UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA

EN INFORMÁTICA DE GESTIÓN

PROYECTO FIN DE CARRERA

PRUEBAS ESTRUCTURALES: FLUJO DE DATOS

Tutor: Manuel Velasco de Diego

Autor: Fco Javier Ordóñez Morales SEPTIEMBRE, 2004

1. INTRODUCCIÓN

1.1. PROBLEMA GENERAL

Existen muchos ejemplos de mal funcionamiento de programas de ordenador, que en algunas ocasiones no tienen una grave importancia, pero en otras pueden ser de gran riesgo al estar en juego la vida de las personas. Entre los más graves podemos encontrar accidentes de aviones y trenes. Y como ejemplos de menor importancia se pueden citar las largas horas de espera en los aeropuertos, en los bancos, etc. El campo de la informática invade, poco a poco, todos los sectores de nuestra sociedad.

La prueba de programas es el conjunto de medidas que se van a tomar a lo largo de las diferentes fases de fabricación de un programa para aumentar las posibilidades de obtener al final un programa que se corresponda con los objetivos de utilización deseados. La prueba forma parte del proceso de desarrollo y se interesa únicamente por el producto final de la actividad de los programadores (las personas que han desarrollado el programa), el texto del programa (el código fuente) y por el comportamiento que presentará.

Básicamente hay dos tipos de técnicas de prueba de programas que son las Pruebas Funcionales (caja negra) y las Pruebas Estructurales (caja blanca). La fase de prueba es la que consume mayor parte de los recursos humanos o materiales (50%, incluso el 60% del coste total del proyecto, considerando aparte los costes del mantenimiento). Por ello, hay que seleccionar entre una de estas pruebas la más conveniente. La solución nunca será realizar menos pruebas, sino hacerlas mejor, es decir, escoger la estrategia y la técnica de prueba más adecuadas que sean capaces de detectar el máximo número de defectos y anomalías reales, entendiendo por "reales" las que se producen en los procesos de desarrollo.

1.2. PROBLEMA ESPECIFICO

El desarrollo de este proyecto es crear una aplicación que pruebe programas observando el comportamiento del código fuente y minimice las probabilidades de aparición de una anomalía durante el desarrollo del programa.

La técnica de prueba desarrollada en el proyecto es el análisis del flujo de datos. Esta técnica analiza el flujo de las variables de un programa tal como es definido en orden secuencial de las diferentes definiciones y usos. Intenta encontrar anomalías en el flujo de datos, para ello construye las DR-Cadenas de todas las variables que aparecen en el programa.

Existen tres anomalías que se pueden considerar como generales, pero también existen otras correspondientes a las estructuras (*if, while, do*...). A continuación se describen las tres anomalías:

1) **r...** la variable tiene un valor indefinido durante su primera utilización
2) **...dd...** dos definiciones consecutivas, de las cuales la primera no es necesaria.
3) **...d** última definición innecesaria.

En cuanto a los parámetros se puede comentar lo siguiente:

- Las variables pasadas por valor deben tener una cadena de la forma r...r
- Las variables pasadas por referencia deben tener una cadena ...d...
- Las variables locales deben atenerse a las anomalías generales, teniendo en cuenta que la anomalía consistente en dos definiciones consecutivas debe aplicarse también a las variables por valor y referencia.
- Las constantes deben tener una cadena formada solo por referencias.

1.3. TERMINOLOGÍA

Anomalía

Comportamiento observado que se diferencia del comportamiento esperado o especificado.

Prueba de programas

Proceso perteneciente a la garantía de calidad del software, que se encarga del estudio de un programa, ya sea a través de su ejecución u observando el código, con la intención de descubrir errores.

Estrategia de prueba

Análisis de requisitos del software, dónde se establece el campo de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software.

Flujo de datos

Caminos que pueden tomar las variables pertenecientes a un programa a lo largo de las estructuras y funciones del mismo.

Flujo de control

Camino real que toma cierta variable durante la ejecución de un programa.

Análisis del flujo de datos

Técnica de prueba basada en el minucioso examen de los detalles procedimentales y que comprueba los caminos lógicos del software, de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.

DR-Cadena

Valores que pueden tomar las variables durante el flujo de datos de un programa.

Datos de prueba (DP)

Entradas que se proporcionarán al programa durante una ejecución.

1.4. ESTRUCTURA DEL DOCUMENTO

Se ha procurado estructurar este proyecto de fin de carrera de forma que resulte lo más fácil posible introducir al lector en los conceptos básicos del filtrado de información. Para ello se ha seguido un orden lógico. Antes de usar los términos y conceptos, éstos se definen en la primera parte del proyecto.

- **Capítulo 1:** El primer bloque del proyecto está compuesto por cuatro apartados que tienen como objetivo introducir al lector en los conceptos básicos de la documentación. Se introduce la prueba de programas en general y la técnica de prueba desarrollada en el proyecto.

- **Capítulo 2:** En este capítulo se ilustrarán los aspectos teóricos sobre los cuales se desarrolla este proyecto. Se realiza un resumen de las pruebas de programas que existen y sus tipos, explicando en detalle la técnica del "Análisis del flujo de datos", con algunos ejemplos aclarativos.

- **Capítulo 3:** En el tercer capítulo se muestra una visión general amplia de los objetivos que se pretenden alcanzar con el desarrollo de este proyecto.

- **Capítulo 4:** En el presente capítulo se inicia un bloque de apartados dedicados a estudiar el entorno de trabajo del proyecto, se explican las herramientas utilizadas que son básicamente el lenguaje de programación C++ y el editor visual sobre windows de este lenguaje, Visual C++. Se resumen sus utilidades y algunas características.

- **Capítulo 5:** El capítulo quinto es el más importante en este trabajo. Profundiza en el desarrollo del trabajo realizado mostrando de forma amplia y rigurosa las estructuras de datos, las funciones y el funcionamiento general del programa.

- **Capítulo 6:** El sexto capítulo presenta unos ejemplos de ejecución, algunos sencillos y otros más complejos, tal como podría realizarse en papel y cómo son presentados por el programa que se ha desarrollado.

- **Capítulo 7:** En el presente capítulo se presentan los resultados y las conclusiones más importantes obtenidas.

- **Capítulo 8:** En este último capítulo del proyecto se realizan una serie de propuestas para la continuación del mismo.

2. ESTADO DE LA CUESTIÓN

2.1. PRUEBAS DE SOFTWARE

En este apartado se hace una presentación de la fase de pruebas dentro del ciclo de vida software, destacando su importancia, presentando las diferentes técnicas, excepto la técnica de flujo de datos que se comenta en el apartado siguiente.

Existen casos en la vida real que demuestran que el software en producción tiene en numerosos casos anomalías fatales. Ejemplos conocidos y graves son, por ejemplo, los fallos ocurridos con el control de radiaciones en un acelerador de partículas en el hospital de Zaragoza, el grave accidente de trenes de alta velocidad en Alemania, numerosos accidentes de aviones con fallos en los elementos de medición, o la explosión nada más despegar de la nave espacial Challenger. Todos estos casos sensibilizaron a la opinión pública, y todos eran fallos de software.

Existen otros casos menos graves, que no por ello deben obviarse, como pueden ser las largas horas de espera de los viajeros en los aeropuertos, los errores en las cuentas bancarias, etc. Ocurrió un caso curioso en Estados Unidos, en el departamento de Hacienda. Se solicitó a un ciudadano, por vía judicial, que pagara sus impuestos relativos al año en curso, con un total de cero dólares. El programa que calcula el importe del pago no tuvo en cuenta que el valor no era positivo y automáticamente, al no pagar el interesado, se tomaron medidas innecesarias.

Puede decirse con estos ejemplos que las pruebas del software no deben seguir siendo un asunto de algunos especialistas, pues los programas están cada vez más y más implicados en nuestra vida cotidiana.

2.1.1. Definición de prueba

Una actividad de gran importancia es la garantía de calidad, es decir, el conjunto de medidas que se van a tomar a lo largo de las diferentes fases de fabricación de un

programa para aumentar las posibilidades de obtener al final del proceso un programa que se corresponda con los objetivos de utilización deseados.

La prueba no engloba todas las actividades relativas a la garantía de la calidad, pues no se ocupa de las diferentes etapas que han precedido al desarrollo del programa (cómo se han realizado las especificaciones, con qué formalismo, cómo se ha escrito el manual de usuario, qué elecciones se han hecho para la concepción de la arquitectura, etc.).

La prueba tiene por objetivo minimizar las probabilidades de aparición de una anomalía durante la utilización del programa.

Anomalía es un comportamiento observado que se diferencia del comportamiento esperado o especificado. La anomalía es producida porque existe un error en el software.

Existe una cadena causal: error ---> defecto ---> anomalía.

Dado un error no necesariamente aparece una anomalía, debido a que puede que la ejecución del software no tenga en cuenta la parte de código que contiene el error. Pero si es necesario, que dada una anomalía, debe existir un error que la produce. Es similar al cuerpo humano. Se puede tener un virus (error) en el organismo que produzca o no una enfermedad (anomalía).

El conjunto de errores o defectos que puede afectar a un programa es, por supuesto, infinito, y es imposible darles una calificación exacta. Se pueden identificar seis clases de errores:

- Cálculo: existencia de una instrucción x:=x+2; en lugar de x:=y+2.
- Lógica: mala expresión de un predicado. "IF (a>b) THEN" en lugar de "IF (a<b) THEN".

- Entrada/Salida: Defectos que expresan una mala descripción, una mala conversión o un formato inadecuado de las entradas/salidas, en la comunicación con el exterior.
- Tratamiento de datos: Mal acceso o manipulación de datos, mala utilización de un puntero, variable no definida, desbordamiento del índice de una tabla, etc.
- Interfaz: Mala comunicación entre los componentes internos del programa, por ejemplo, que se llame al procedimiento P1 en lugar de llamar al procedimiento P2, que el paso de parámetros no sea correcto, etc.
- Definición de datos: Un dato ha sido declarado como entero en lugar de real, un valor es de precisión simple en lugar de serlo de doble, etc.

Puede definirse prueba de un programa como una actividad que forma parte del proceso de desarrollo. Se lleva a cabo según las reglas de la garantía de la calidad y comienza una vez que la actividad de la programación ha terminado. También se interesa tanto del código fuente como del comportamiento del programa. Su objetivo consiste en minimizar las probabilidades de aparición de una anomalía con los medios automáticos o manuales que vienen a detectar también, tanto las diversas anomalías posibles, como los eventuales defectos que las provocarían.

La prueba es una actividad que, por su esencia, no podrá jamás demostrar la exactitud de ningún programa. No existe ningún método algorítmico, y con mayor motivo ninguna herramienta automática, que sea capaz de afirmar de una manera formal y general, la exactitud de un algoritmo dado.

Hay que esforzarse en detectar el mayor número de anomalías o de defectos. Se considera que una prueba que ha tenido éxito no es una prueba que no ha encontrado ningún defecto, sino una prueba que ha encontrado efectivamente un defecto, o bien una anomalía.

2.1.2. Dificultad de la prueba

La fase de prueba consume la mayor parte de los recursos humanos o materiales, un 50 %, incluso el 60 % del coste total del proyecto, considerando aparte los costes del mantenimiento.

Para realizar mejor la prueba, es necesario acotar todos los parámetros que hacen que la prueba sea difícil. Estos factores pueden clasificarse de la siguiente forma:

- Dificultades asociadas a los procesos de introducción de defectos:

El proceso de desarrollo de un programa es un proceso de transformaciones sucesivas. Estas transformaciones sucesivas no conciernen directamente a un objeto físico, como podría ocurrir en un producto industrial cualquiera, sino a un conjunto de informaciones.

Las ideas abstractas y "confusas" de un usuario se transforman primero en las especificaciones. Éstas se transforman en el diseño, posteriormente en el código fuente, etc.

Estas traducciones sucesivas provocan pérdida de información. Además, el gran número de etapas de transformación hace que una anomalía detectada en una fase terminal pueda ser el resultado de varios pequeños deterioros anteriores. Las consecuencias serán más graves cuanto más tiempo pase entre el momento en el que el error es cometido y aquél en que el defecto es detectado.

- Dificultades de orden psicológico o "cultural":

La razón principal es que se entiende en muchos casos este proceso como dotado de una componente destructiva. Existen otras componentes culturales, como pueda ser la falta de interés e infravaloración. También se piensa equivocadamente

que los errores costosos aparecen en las fases iniciales, que es donde se debería centrar principalmente el proceso de prueba.

- Dificultades formales:

Existe un teorema demostrado que dice que no puede existir ningún algoritmo general (ni por lo tanto ninguna herramienta) capaz de demostrar la exactitud total de cualquier programa.

A la vez, existen algoritmos imposibles, como el siguiente: "Escribir un algoritmo, que sin utilizar ficheros de copia secundaria, u otros artificios de este tipo, dé como resultado de ejecución la impresión de su texto fuente".

Si nadie experimenta el deseo de probar un programa, y si por añadidura no existe y no puede existir ninguna herramienta capaz de asegurar su puesta en funcionamiento automática, ¿debería concluirse por tanto que la imposibilidad de prueba de los programas informáticos es algo irremediable? Por supuesto, la respuesta es NO. Existen numerosas ideas que intentan apoyar el proceso de prueba.

En todo acto creativo existe un deseo de perfección. El programador debe ser la primera persona que prueba su algoritmo.

Es cierto que es imposible poner en marcha un algoritmo general de prueba, pero se pueden desarrollar herramientas específicas.

Existen dos axiomas, que ayudan a implantar psicológicamente el proceso de prueba:

- El axioma del programador competente: El programa que ha proporcionado un programador se parece sensiblemente al programa teóricamente correcto que debería haber escrito, y que por definición, no podemos conocer.

- El axioma de acoplamiento: Pequeñas causas pueden producir grandes efectos, es decir, en términos de prueba de programas, las anomalías complejas están siempre provocadas por una combinación (acoplamiento) de defectos simples.

2.1.3. Conceptos fundamentales

La actividad de prueba consiste en buscar de una manera estática los defectos simples y frecuentes, o definir las entradas que se proporcionarán al programa durante una ejecución. Estas entradas se denominan "datos de la prueba" (DP).

El conjunto de datos de prueba que se produce para la prueba será denominado "juegos de prueba". Para poder aplicar estos juegos de prueba, debería efectuarse una secuencia de acciones. Esta secuencia de acciones se denomina "escenario de prueba". Los datos de prueba deben constituir una muestra representativa de todas las entradas posibles que pueden someterse al programa. Producir todas las entradas posibles sería lo que se denomina prueba exhaustiva.

Un juego de prueba adecuado es un conjunto de DP que es capaz de diferenciar el programa bajo prueba de otra versión del programa que sea errónea.

Las técnicas de prueba que producen sus juegos de prueba basándose en los resultados obtenidos por la ejecución del programa, sin preocuparse por la estructura interna del programa, se denominan "funcionales". Suelen denominarse técnicas de "caja negra", pues se considera el programa como una caja negra y opaca cuya estructura interna (es decir, el código fuente) no se puede ver.

Las técnicas que producen sus juegos de prueba analizando el código fuente se denominan técnicas de prueba de "caja blanca" o bien "estructurales". En este caso, el programa es, pues, considerado como una caja blanca que deja ver su estructura interna.

Las técnicas de prueba pueden ser también clasificadas según se ejecute o no el código objeto. Las técnicas que ejecutan el código objeto examinando el

comportamiento real del programa se denominan también técnicas de prueba "dinámicas". Las otras técnicas que examinan la forma pasiva del programa, sin ejecutarlo, son denominadas "estáticas".

Como ejemplo ilustrativo de esta clasificación puede indicarse el de un mecánico probando un coche. La prueba funcional consistiría en conducir el coche para ver si la velocidad desarrollada es la indicada por las especificaciones, o frenar sobre una carretera mojada para verificar el buen funcionamiento de los frenos. La prueba estructural consistiría en examinar el motor en marcha (dinámica) y parado (estática).

Las técnicas de prueba de programas adoptan la misma filosofía complementaria (combinación de técnicas funcionales, estructurales, dinámicas y estáticas). Hablamos, en este caso, de prueba "caja gris", consistente en crear en primer lugar las pruebas funcionales para examinar a continuación las partes del código fuente que quedan por estudiar estructuralmente.

Existe una paradoja: las técnicas más utilizadas son las técnicas funcionales, mientras que las técnicas que han generado mas interés por parte de los investigadores (y por tanto de un gran volumen de resultados teóricos) son las estructurales.

2.1.4. Nociones de estrategia de prueba

Las técnicas de prueba deben formar parte de una estrategia global de prueba. La definición de una estrategia de prueba lleva consigo:

- La definición de recursos puestos en marcha, tales como equipos, responsables de pruebas, herramientas, etc.

- Los mecanismos del proceso de prueba, como la gestión de las configuraciones, las reuniones durante las cuales se evaluarán el proceso de prueba y sus resultados, etc.

Deben tenerse en cuenta diversos factores: los métodos de especificación y diseño utilizados, la experiencia de los programadores, la epidemiología de los defectos corregidos, el lenguaje de programación adoptado (lenguaje clásico, orientado a objetos, etc.), el tipo de aplicación realizado (sistemas de información, programas de tiempo real, sistemas basados en conocimiento, etc.).

Si se prueban independientemente los diferentes elementos que producen el programa, se hablará de prueba unitaria. Cuando lo que importe sea la disposición de estos elementos y la forma en que se comunican entre sí, se hablará de prueba de integración.

Si la intención es asegurar que el producto se está desarrollando de una forma correcta y que los diferentes elementos que lo componen no son deficientes, se hablará de verificación para designar el punto de vista del ingeniero de software.

Un programa no es un fin en sí. Solamente tiene razón de ser si es útil, es decir, si responde a las necesidades de un usuario final. Un componente primordial de una estrategia de prueba es justamente asegurarse esta buena adecuación a un nivel de confianza aceptable. Para designar esta problemática, se adoptará el término validación.

Sin duda la distinción entre validación y verificación puede concretarse mejor según una fórmula bien conocida:

- Se verifica que se ha hecho bien un programa.
- Se valida que se ha hecho un buen programa.

Verificación y validación son pues complementarias.

2.2. CLASIFICACIÓN DE LAS TÉCNICAS DE PRUEBA

Existen varias técnicas de prueba. Cada técnica afrontará el proceso de prueba desde una perspectiva diferente, esperando detectar y cubrir un cierto conjunto de clases de errores. Se presenta a continuación una clasificación práctica de las técnicas de prueba. Esta clasificación se realiza en función de los elementos necesarios para llevarla a cabo. Para aplicar una técnica, se necesitan, al menos, uno de los siguientes elementos:

- Las especificaciones del programa
- El código fuente del programa
- El código binario ejecutable correspondiente

	Objetos necesarios			**Ejemplos de Técnicas**
Clase	Especificaciones	Código Fuente	Código Objeto	
1				Ciertas pruebas aleatorias
2				Análisis estático del flujo de datos, Revisiones de código, Cálculo de complejidad
3				Análisis dinámico del flujo de datos
4				Análisis de coherencia de especificaciones formales
5				Pruebas en los límites, Grafos causa-efecto, Pruebas particionales
6				Ejecución simbólica, Prueba formal
7				Métodos de cobertura, Pruebas mutacionales

Tabla 2.1: Clasificación de las técnicas de prueba

La clasificación se basa en la combinación de estos 3 objetos. Enumerando las 7 combinaciones posibles, se forman 7 clases, mostradas en la tabla 2.1, del 1 al 7.

Cada clase contiene un subconjunto de las técnicas de prueba:

- Clase 1: Se agrupan en esta clase todas las técnicas de prueba ciegas (pruebas hechas por el usuario o un cierto conjunto de pruebas aleatorias, etc.).
- Clase 2: Se examina el código fuente para identificar errores universales, es decir, independientes de toda aplicación. Pertenecen a esta clase las técnicas de análisis estático del flujo de datos, las revisiones de código (por ejemplo, verificación de los estándares), los analizadores de complejidad, etc.
- Clase 3: Esta clase está formada por el análisis dinámico de los datos.
- Clase 4: Están comprendidas en esta clase todas las técnicas de análisis de las especificaciones.
- Clase 5: Se trata de un conjunto de técnicas muy extendidas: pruebas en los límites, grafos causa-efecto, la mayor parte de las pruebas aleatorias, pruebas basadas en la sintaxis, etc.
- Clase 6: Pertenecen a esta clase las técnicas de ejecución simbólica del programa y las técnicas de prueba de exactitud del algoritmo.
- Clase 7: Aquí están agrupadas la mayor parte de las técnicas estructurales: cobertura de las instrucciones, de los arcos del grafo de control, etc. Pertenecen también a este grupo las técnicas de pruebas mutacionales (débiles o fuertes) y los métodos de cobertura basados en el flujo de datos.

Esta clasificación no es la única posible. Las técnicas que no necesitan el código fuente (clases 1, 4 y 5) se denominan técnicas funcionales, o bien "caja negra", pues prueban la funcionalidad del software especificado.

Las demás técnicas (clases 2, 3, 6 y 7) se denominan técnicas de prueba estructurales, o "caja blanca". Crean los datos de prueba basándose en el código fuente.

Las técnicas de prueba pueden también agruparse según otro criterio: la ejecución o no del código binario correspondiente. Las técnicas que ejecutan el código binario examinando el comportamiento real del programa se denominan dinámicas (clases 1, 3, 5 y 7). Las otras técnicas que examinan la forma pasiva del programa (el código fuente), son las denominadas estáticas (clases 2, 4 y 6).

Cada técnica de prueba tiene sus ventajas y sus inconvenientes. Los inconvenientes de una categoría pueden ser las ventajas de otra y a la inversa.

Como ejemplo puede presentarse el siguiente fragmento de código, implementado en pseudolenguaje Pascal:

```
function sum (x,y: integer) : integer;
begin
     if (x = 600) and (y = 500) then  sum := x-y
     else                             sum := x+y;
end
```

Se supone que el fragmento de código anterior representa una función que dados dos números enteros de entrada realiza la suma de ambos y la devuelve como resultado de la función. Puede observarse que existe un error, ya que en el caso de que los valores de entrada sean, respectivamente, 600 y 500, el valor devuelto no es la suma de los valores sino la resta. En este caso las técnicas funcionales no serán capaces de detectar el error y las técnicas estructurales pueden detectarlo, además de emplazarlo exactamente en el código.

2.2.1. Eficacia de las técnicas de prueba

En la eficacia de las técnicas de prueba intervienen varios factores:

- La suerte.
- El tipo de aplicación.
- La experiencia del examinador.

- La dificultad de aplicación de la técnica y su rendimiento (tiempo de aprendizaje, precio de la herramienta, etc.).
- El entorno material.
- El tipo de error investigado, su frecuencia, su gravedad.
- El método de desarrollo seguido y la fase en la que se encuentra.
- El lenguaje de programación utilizado.
- La documentación.
- La precisión de localización de la técnica.

Este último factor, la precisión de localización, es muy importante y frecuentemente olvidado. Lo que interesa no es solamente la detección del error, sino también la corrección del fallo responsable. La mayor parte de las técnicas estructurales son capaces de localizar el emplazamiento preciso (en el código fuente) del fallo. Por el contrario, la mayor parte de las técnicas funcionales encuentran el error pero hay que considerar el tiempo necesario de depuración para localizar el fallo. La tabla 2.2 presenta la capacidad de encontrar los errores de algunos tipos de técnicas según los tipos de fallo que existen.

Tipo de Fallo	**Técnica**				
	Revisiones de código	Análisis estático	Prueba de exactitud	Pruebas estructurales	Pruebas funcionales
Cálculo	Media	Media	Grande	Grande	Media
Lógica	Media	Media	Grande	Grande	Media
E/S	Grande	Media	Limitada	Media	Grande
Tratamiento de datos	Grande	Grande	Media	Limitada	Grande
Interfaz	Grande	Grande	Limitada	Grande	Media
Definición de datos	Media	Media	Media	Limitada	Media

Tabla 2.2: Capacidad de encontrar errores

Se puede delimitar la noción de eficacia de una técnica comparando los fallos que ha encontrado con los fallos restantes (suponiendo por supuesto que es posible conocer la cantidad de estos últimos). Se calcula mediante la relación de detección, δ.

$$\delta = \frac{\text{número de fallos encontrado}}{\text{número total de fallos}}$$

A partir de las estadísticas efectuadas por ciertas técnicas de prueba, se dispone de una aproximación bastante tosca de la relación de detección:

	Rango de δ
Revisión de código	0,35 - 0,90
Análisis estático	0,20 - 0,40
Pruebas de exactitud	0,50 - 1,00
Pruebas estructurales	0,20 - 0,70
Pruebas funcionales	0,20 - 0,70

Puede decirse que lo que caracteriza una buena técnica de prueba, no es su eficacia para la detección de errores en general, sino la posibilidad de localizar errores frecuentes y graves, que son los que proporcionarán un mayor número de problemas.

2.2.2. Pruebas funcionales

La prueba funcional no examina el contenido del programa sino su comportamiento funcional. Intenta determinar cuáles son los datos de prueba a partir de este comportamiento funcional. No es posible enumerar todas las combinaciones posibles de DP. Las distintas técnicas difieren entonces en los criterios de selección de los DP, basándose en las especificaciones.

Dos métodos diferentes pueden producir DP semejantes y el mismo método puede producir DP diferentes, dependiendo de numerosos factores (examinador, generador de DPs, etc.).

Entre las técnicas que se agrupan bajo el nombre de prueba funcional se encuentran las siguientes:

2.2.2.1 Análisis particional

El objetivo de cada programa es el tratamiento de los datos de entrada. Tipos de datos diferentes implican diferentes procesos, en función de las especificaciones. La división de los datos trae consigo el establecimiento de clases diferentes, lo que implica una partición de estos datos en diferentes clases.

Para construir cada una de las clases de equivalencia se selecciona un representante de cada clase, siendo éste cualquiera que pertenezca a la clase. Cada representante formará un dato de prueba.

Para construir las clases se examinan:

- Los campos de valores E/S
- El conjunto de funciones a realizar por el programa

Las clases se forman del siguiente modo:

- Regla 1: Si el dato es un intervalo:
  - Valores inferiores, si son posibles
  - Valores superiores, si son posibles
  - N clases válidas

- Regla 2: Si el dato está formado por un cierto número de valores:
  - Número menor de valores
  - Exceso de valores
  - N clases válidas

- Regla 3: Si el dato es un conjunto de valores tratados de forma diferente:
  - Clase válida para cada valor válido

- Clase no válida para los demás

- Regla 4: Si el dato es una obligación o requisito: (forma, sintaxis, sentido)
  - Clase con el requisito respetado
  - Clase con el requisito no respetado

Las clases válidas sólo se tendrán en cuenta una sola vez, no repitiendo valores. Lo mismo sucede en el caso de clases no válidas obtenidas por medio de opciones diferentes.

Para todas las variables a estudiar se aplicarán las reglas 2 y 4, y dependiendo del tipo de datos, la regla 1 o la regla 3.

2.2.2.2. Prueba de los límites

Es uno de los métodos funcionales más eficaces. Los errores aparecen en los casos en los que no se había previsto el comportamiento de un programa para valores límites:

- Valores muy elevados
- Valores nulos de un índice
- Valores negativos o máximos del índice de un bucle
- Incluso datos no válidos

No sería nunca seguro, por ejemplo, un avión que ha sido probado únicamente en condiciones de meteorología ideales. Para probar eficientemente un avión, se han de reproducir unas condiciones límites (altas temperaturas, avería simulada de un reactor, entrada de un pájaro en un reactor, etc.). Si hay un problema, no se reproducirá jamás en condiciones normales, pero hay grandes posibilidades de que aparezca en condiciones extremas.

La prueba de los límites comprende valores justo antes, en, y justo después de los límites de definición. Para realizarla hay que identificar límites inferiores y superiores para cada variable de E/S:

- Si la variable esta comprendida en un intervalo de valores
  Se toma la menor variación posible DELTA
  Se generan 6 valores
    - 2 iguales a los límites
    - 4 valores = valores de los límites ± DELTA

  Si los valores son enteros, DELTA es entonces igual a 1. Si son reales puede tomarse por convención un valor fijo, siempre que éste no pueda deducirse matemáticamente.

- Si la variable es un conjunto ordenado de valores
  Se elige el primero, el segundo, el penúltimo y el último elemento.

La noción de límite no es siempre evidente y, por lo tanto, se estudian otros casos. Por ejemplo, si se define el intervalo [-10, 10] para dos variables a y b, se suelen tomar valores límite tales como

a = b
a o b = 0

Los límites son el lugar preferido por la mayor parte de los errores, cubriendo toda la gama de las fases de la prueba: unitaria, integración, instalación y de sistema.

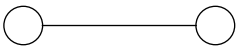
2.2.2.3. Grafos causa efecto

Consiste en construir una red que una los efectos de un programa (salidas) a las causas (entradas) que están en su origen.

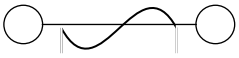
Esta red está compuesta por nodos y segmentos. Los segmentos van a simbolizar las uniones lógicas. Para elaborar la red se parte del análisis de las especificaciones.

Existen cuatro tipos de símbolos para formar el grafo de causa-efecto.

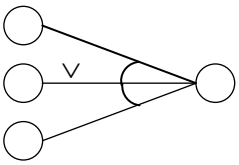
IDENTIDAD



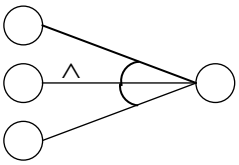
NEGACIÓN



Ó LÓGICO



Y LÓGICO



La red formada se traducirá mediante una tabla de decisión (tabla de verdad). En esta tabla de verdad, el valor verdadero está representado por el 1 y el valor falso por el 0.

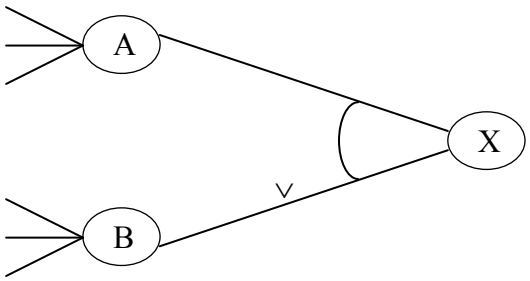
Con la tabla de verdad se obtienen las salidas a partir de las entradas navegando por el grafo mediante las reglas lógicas.

El valor de un efecto se obtiene a partir de los valores de sus causas.

El problema surge cuando las causas se hacen cada vez más numerosas. En este caso el tamaño de los DP crece de forma exponencial.

Hay que definir reglas que ayuden a limitar el número de DP a probar. Para ello se establece la siguiente forma de trabajo:

- Para un grafo causa-efecto bajo forma **disyuntiva**

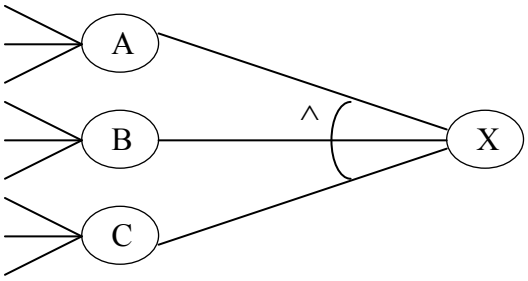


Se tienen las reglas de construcción siguientes:

R1: Si X debe estar en el estado 1, no tener en cuenta la situación A=B=1. Sólo se tendrá en cuenta aquellas situaciones en las que, siendo verdaderas, haya al menos un 0. Deben seguir aplicándose reglas.

R2: Si X debe estar en el estado 0, enumerar **todas** las situaciones A=B=0. Al decir todas, se refiere a valores anteriores de los nodos para el caso en que haya varios niveles.

- Para un grafo bajo forma **conjuntiva**



Se tienen las reglas simplificadoras siguientes:

R3: Si X debe estar en el estado 1, enumerar **todas** las situaciones donde A=B=C=1.

R4: Si X debe estar en el estado 0, entonces

R.4.1. No incluir más que **una** situación donde A=B=C=0

R.4.2. En las demás combinaciones de A, B y C (es decir 001, 010, 100, 011, 101, 110), cuando exista un 1 para un estado (por ejemplo en la combinación 101 los estados A y C son puestos a 1), bastará escoger **una** única muestra de causas dándole a este valor 1, tomando **todas** las del caso 0.

La técnica de prueba basada en los grafos causa-efecto puede resumirse mediante las etapas siguientes:

- Si el objeto probado tiene numerosas funcionalidades, descomponerlo en entidades más sencillas.
- Identificar las causas (condiciones de entrada o clase de condiciones de entrada).
- Identificar los efectos (condiciones de salida o transformación del sistema).
- Establecer un grafo de las relaciones entre causas y efectos.
- Completar el grafo mediante los contratiempos entre causas o entre efectos.
- Convertir el grafo en tabla de decisión:
    - Poner sucesivamente en Verdadero (o Falso, según el caso que se quiera probar) todos los efectos.
    - Crear una línea para cada combinación de causas que producen el efecto.
    - Para cada combinación, buscar también el estado de todos los efectos.
- Producir un DP por línea tras la simplificación de la tabla según las reglas estudiadas.

Este método es muy completo y preciso. Es posible automatizar algunas de sus fases.

El mayor problema consiste en que los grafos pueden llegar a complicarse demasiado, sobre todo en el caso en el que el número de causas sea elevado. En estos casos suelen añadirse nodos intermedios para representar las combinaciones lógicas de las diferentes causas.

Al modificarse las especificaciones puede tener que variarse el grafo. Al añadirse nuevos nodos, ya sea nodos causa o nodos efecto, debe redefinirse la estructura intermedia.

2.2.2.4. Pruebas aleatorias

En los tipos anteriores de pruebas funcionales la selección de los datos de prueba se hacía de un modo determinista. En las pruebas aleatorias la generación de DPs se hace de una manera probabilística: en la mayoría de los casos mediante un muestreo uniforme de los parámetros de entrada del programa.

Ventajas de esta técnica:

- La generación de los DP puede ser fácilmente automatizada.
- Objetividad de los DP producidos: existe, sin embargo, dependencia de la calidad del generador de números aleatorios (más bien pseudo-aleatorios) utilizado.

Inconvenientes:

- Dificultad de producir al azar combinaciones de entradas que produzcan comportamientos muy específicos:

  If x=5 and y=7 then ...

Se ha demostrado que las aproximaciones aleatorias son al menos un 80% tan correctas como las aproximaciones deterministas.

Sin embargo, los resultados de una prueba aleatoria tienen siempre un límite que no puede mejorarse, al contrario que las deterministas, en las que un mayor esfuerzo en la prueba puede conducir a resultados sensiblemente superiores.

Para mejorar las pruebas aleatorias, por ejemplo, se estudian las distribuciones de los datos de entrada, es decir, para no ceñirse a su "posible" uniformidad.

El problema es que esta técnica va en contra de la idea de pruebas en los límites.

Otra aproximación consiste en la utilización de pruebas estadísticas estructurales, que analizarían el código, viendo la probabilidad que tiene cada sentencia de ejecutarse.

2.2.2.5. Análisis transaccional

Es una técnica que consiste en modelizar el comportamiento funcional de un programa mediante un conjunto de primitivas funcionales.

La representación se realiza en forma de grafo. La estrategia de prueba está basada, por tanto, en la cobertura del grafo (hasta ahora utilizable solo en técnicas estructurales).

Una transacción es para el usuario una unidad de trabajo y para el sistema es una secuencia de operaciones. El nacimiento de una transacción es el resultado de una acción externa al sistema. El funcionamiento del sistema se representa mediante flujos de transacciones (dibujados mediante grafos). Las transacciones son diferentes unas de otras, pero poseen operaciones en común.

Es la técnica funcional por excelencia ya que solo se deben identificar las transacciones y las acciones del sistema concerniente.

Los grafos de flujos de transacciones son tipos especiales de diagramas de flujos de datos (DFD). Todos los tratamientos están identificados y representados.

El grafo transaccional debe estar bien estructurado para poder probarlo bien. Debe prestarse especial atención a los puntos de unión, de separación, etc.

Una vez obtenido el grafo transaccional, ¿qué se debe probar?

- Caminos suplementarios para tratar los bucles, los valores límites, etc.
- Caminos suplementarios para los casos difíciles, las transacciones largas, complejas, arriesgadas.
- Unos casos de pruebas suplementarios para probar las creaciones, fusiones, separaciones y desapariciones de las transacciones.

El conjunto de caminos de prueba se buscaría de un modo similar a los grafos de control en las pruebas estructurales.

Se pueden emplear los métodos de cobertura en caminos.

2.2.2.6. Análisis sintáctico

El análisis sintáctico es una prueba funcional que consiste en determinar los diferentes datos de prueba, de entrada a la aplicación, mediante una descripción formal de estos datos, que viene dada en forma de gramática BNF o autómata en estados finitos.

Es una buena técnica para aplicaciones que necesitan datos de entrada que respetan una sintaxis rígida y bien definida, como puedan ser compiladores, intérpretes de comandos, diálogos de operador, relaciones intercalculadoras, encadenamiento de menús, etc.

Para realizar esta técnica se definen 3 fases bien diferenciadas:

- Definir la gramática.

La gramática debe ser tipo 2 o tipo 3, es decir, independiente del contexto o regular. Las gramáticas tipo 0 ó 1 dan muchos problemas a la hora de realizar los siguientes pasos. Es necesario que en todas las producciones haya un único símbolo no terminal a la izquierda de la producción. De igual forma, es necesario que no existan símbolos lambda.

- Construir el árbol de derivación de la gramática.

Se construye a partir de la gramática obtenida en la primera fase. Cada símbolo, ya sea terminal o no terminal se incluirá en un nodo distinto. Los nodos se numeran a partir del número 1. En el árbol deben distinguirse los distintos niveles y en éstos si los nodos son terminales o no terminales.

- Obtener los datos de prueba, mediante división en comandos válidos y comandos no válidos.

Se obtienen a partir del árbol, dividiéndolos en dos partes: comandos válidos y comandos no válidos. Esta división se realiza para agrupar los distintos mensajes, ya sean de error o no.

Respecto a los comandos válidos: es necesario obtener DP representativos de todos los nodos.

Respecto a los comandos no válidos: Existe un gran número de ellos y al no poder seleccionarse todos se tomará una muestra suficientemente representativa.

Como resumen de comandos no válidos, en particular para cada nodo, podría decirse lo siguiente:

- Si el nodo es no terminal se procede a su omisión.
- Si el nodo es terminal se procede también a su modificación.

Esta técnica de prueba funcional está particularmente adaptada a aplicaciones que tienen una sintaxis de entradas rígida. Es pues fácilmente automatizable pero necesita una buena "compartimentación" de las diferentes sintaxis si no se desea afrontar una fuerte explosión combinatoria.

2.2.3. Pruebas estructurales

La prueba estructural va a producir los DP o sus conclusiones cualitativas basándose en la estructura del código fuente. Existen dos tipos: estáticas y dinámicas. Las técnicas estáticas son las más estudiadas. Los métodos estáticos son todos los métodos estructurales que no necesitan de la ejecución del código fuente para poder realizarse.

Los errores van a interpretarse de un modo universal, ya que son comunes a todos los lenguajes de programación.

No está considerado el aspecto práctico de la obtención de los DP, ya que este tipo de análisis tiene un componente estrictamente formal.

Entre ellas se encuentran las siguientes, a excepción de la prueba de análisis del flujo de datos que se encuentra detallada en el siguiente apartado:

2.2.3.1. Modelo de McCabe

El número de errores existentes en un programa está directamente relacionado con su complejidad. Existen modelos matemáticos para medir la complejidad.

Este modo de obtener la complejidad se llama también complejidad ciclomática. Se basa en la estructura del grafo de control asociado a cada programa.

Hay que calcular el número de McCabe, v(G), que es igual al número de circuitos independientes que formaban la base, y que también es igual al número de condiciones simples existentes más uno.

La complejidad de un programa está directamente ligada a las diferentes ejecuciones posibles (caminos) que pueden resultar.

Se utiliza también la noción de complejidad esencial, denominada ev(G). Este valor es el número de McCabe asociado al más pequeño grafo de control no reducible. Es decir, si el programa está bien estructurado este valor es igual a 1.

Así, la complejidad esencial indica el nivel de no estructuración de un grafo de control.

Se propone el valor 10 como límite "máximo" para el valor de v(G).

2.2.3.2. Modelo de Halstead

Para calcular la complejidad de un algoritmo este modelo tiene en cuenta la distribución de las instrucciones y variables que pone en juego el programa. Así, se estimará a partir del número y frecuencia de las instrucciones y términos que aparecen en el programa.

La complejidad se calculará a partir de 4 cantidades:

n1: número de operadores diferentes que aparecen en el programa.

n2: número de operandos (términos) diferentes (constantes o variables).

N1: número total de operadores del programa.

N2: número total de operandos.

Al escribir la instrucción "a := a + 1 ;" se utilizan tres operadores y tres operandos. Los operadores son la suma ("+"), la asignación (":=") y el fin de instrucción (";"). Los operandos son el símbolo *a* y el *1*.

Un operador se traduce en una palabra clave reservada como read, input, while, etc.

Un término se traduce en una variable (o una constante) sobre la cual se aplica el operador.

Halstead parte de la hipótesis de que, en un algoritmo normal, es necesario, desde el punto de vista estático, que la ecuación siguiente se cumpla:

$N1 + N2 \cong n1 \cdot \log_2(n1) + n2 \cdot \log_2(n2)$

La magnitud N1+N2 se denomina longitud del programa.

Se puede igualmente definir el volumen de un programa, es decir, la cantidad de información vehiculada (en bits):

$V = (N1+N2) \log_2 (n1+n2)$

También será interesante comparar el volumen real V de un programa con el volumen más pequeño posible que pueda tener teóricamente: V*. Se denomina como "nivel L de un programa":

$$L = \frac{V^*}{V}$$

Si, por ejemplo, para la copia de una tabla A en una tabla B, el lenguaje de programación permite escribir:

B = A

se utiliza entonces el algoritmo mínimo y tenemos *V*=V*; el nivel del programa es el mayor posible y es igual a la unidad. Pero, si para realizar esta asignación se debe escribir un algoritmo utilizando un bucle, el volumen real V aumentaría reduciendo así el nivel del programa. El término V* no puede ser evaluado con precisión para todos los tipos de algoritmos. Así, en el modelo de Halstead se propone la estimación siguiente:

$$L = \frac{2n_2}{n_1 N_2}$$

Otro concepto es el de nivel del lenguaje que se calcula mediante la siguiente ecuación = $L^2.V$

En la teoría de Halstead la magnitud:

$$E = \frac{V}{L}$$

que se llama Esfuerzo, traduce por excelencia la complejidad de un programa.

2.2.3.3. Ejecución simbólica

La ejecución simbólica consiste en asignar valores simbólicos a las variables y no valores numéricos. A continuación, el programa es ejecutado según la semántica de cada una de sus instrucciones.

En cada instrucción de decisión (if, while, ..., etc.) se crean dos ramas, en las que se incluyen los requisitos de la decisión tomada. La ejecución simbólica se ejecuta entonces sobre las dos ramas para formar un árbol en el que los nodos terminales contienen los valores simbólicos de las variables para todos los caminos posibles de ejecución.

Los resultados de una ejecución simbólica son, en general, largas y complejas expresiones, que pueden simplificarse.

La ejecución simbólica es capaz de detectar ciertas clases de caminos no ejecutables, es decir, código muerto.

Un problema importante que existe en la ejecución simbólica es la explosión combinatoria que acompaña a toda estructura iterativa.

El número total de iteraciones depende del algoritmo de un modo dinámico. Para decidir este número sin ejecutar el programa se pueden tener en cuenta tres casos:

- Utilizar un invariante de bucle.
- Ejecutar el bucle un número arbitrario de veces.
- Entrar en modo interactivo y preguntar por la información suplementaria.

2.2.3.4. Análisis de campos finitos

Es una técnica que utiliza la idea de generación de DP a partir de diferentes clases de equivalencia vista en la técnica funcional de análisis particional. En este caso dos DP pertenecen a la misma clase si sensibilizan el mismo camino en el grafo de control. Tras obtener las clases se seleccionan representantes de cada clase.

Las clases de equivalencia están representadas bajo la forma de polígonos formados por el conjunto de las ecuaciones o inecuaciones que aparecen en las condiciones del algoritmo. La dimensión de estos polígonos depende del número de parámetros de las condiciones, aunque se ven aquí las limitaciones debidas precisamente al número de dimensiones, sobre todo a la hora de la presentación gráfica de los polígonos.

La ventaja de esta geometrización de los campos formados por los DP es clara, pues es posible localizar los DP, no solamente en el interior de un polígono, sino también en las fronteras de este último, hecho que será de máximo interés, pues normalmente los errores cometidos son de este tipo (llamados errores de campo), y son errores que se presentan en los límites de las rectas que delimitan los polígonos. Un error típico de éstos sería, por ejemplo, escribir x<y en lugar de x<=y.

2.3. ANÁLISIS DEL FLUJO DE DATOS

Esta técnica, perteneciente a los tipos de pruebas estructurales, analiza el flujo de las variables de un programa, tal como es definido por el secuencionamiento de las diferentes definiciones y usos. Por ejemplo:

La instrucción

```
X := a + b
```

dice que la variable X está definida (es decir, que su contenido va a ser probablemente modificado) y las variables a y b están referenciadas (es decir, utilizadas).

Otros ejemplos pueden ser:

read (x)	x está definida.
write (x)	x está referenciada.
if (x=1) then x := 7	x está referenciada y después definida.
a [i] := x	a está definida, i y x están referenciadas.
x := x + 1	x está referenciada y después definida.

En la ejecución de un programa una misma variable puede definirse o utilizarse varias veces. Cada definición de una variable se representa por la letra d y cada referencia por la letra r, formando lo que se llama cadena-Π de la variable o "DR-Cadena", que refleja el flujo de la variable durante la ejecución del programa.

Esta técnica intenta encontrar anomalías en el flujo de datos. Para ello construye las DR-Cadenas de todas las variables que aparecen en el programa.

La DR-Cadena de una variable x se designa con p (x) ó Π (x).

Existen tres tipos diferentes de anomalías respecto a las DR-Cadenas, que se pueden representar por las siguientes expresiones:

DR-CADENA	**ERROR**	**TIPO ERROR**
r.....	La variable tiene un valor indefinido durante su primera utilización	a
.....d.....	Dos definiciones consecutivas, de las cuales la primera es inútil	a
.....d	última definición inútil	c

Figura (2.1)

Estas tres anomalías se consideran como generales, existentes normalmente en código perteneciente al programa principal, pero pueden existir otras que sean propias solamente de código de procedimientos o subrutinas.

El ejemplo siguiente muestra un caso sencillo de construcción y análisis de DR-Cadenas:

```
Program Ejemplo;
Var a,b,c,d: integer;
Var e,f: string;
Begin
      d:= 5;
      c:= e+d;
      Read(b);
      c:= d+f;
      Write(a,c);
      d:= e+5;
      a:= d+f;
      f:= a+b;
      c:= d+b+2*e;
      e:= f+9;
End.
```

Π(a) = rdr, existe una anomalía de tipo a, pues la variable "a" es referenciada sin que haya obtenido ningún valor previamente.

Π(b)= drr, no existe ningún problema.

Π(c)= ddrd, existe una anomalía de tipo b, pues la variable "c" es definida dos veces consecutivas.

Π(d)= drrdrr, no existe ningún problema.

Π(e)= rrrd, existe anomalía del tipo a, ya que la variable es referenciada, y además varias veces, antes de ser definida, por lo que su valor inicial será indeterminado. También existe un problema de tipo c, cuando finaliza con una definición inútil.

Π(f)= rrdr, existe una anomalía del tipo a, ya que la variable es referenciada antes de ser definida.

Cuando se trate de instrucciones de decisión del tipo if, se evalúan dos DR-Cadenas diferentes: una correspondiente al valor TRUE del predicado y la otra al valor FALSE.

Por ejemplo, para las siguientes instrucciones:

```
a:= b;
If (a=1) then b:= 6+4; else a:= c+b;
If (d=1) then b:= a+8; else a:= c+b+3*2;
```

Se obtiene:

Π(a) = (dr ó drd) + (r ó d) = drr ó drd ó drdr ó drdd
Π(b) = (rd ó rr) + (d ó r) = rdd ó rdr ó rrd ó rrr
Π(c) = (r ó 1) + (r ó 1) = 1 ó r ó rr
Π(d) = r

donde 1, en el caso de la cadena-Π de c, significa que la variable no es referenciada ni definida.

2.3.1. Instrucciones iterativas

Para una instrucción iterativa, se deben producir todas las DR-Cadenas salidas de su ejecución. Por ejemplo:

```
var:= 0;
i:= 1;
While (i ≤ N) do
Begin
     var:= var*i;
     i:= i+1;
End.
```

Se obtiene lo siguiente:

Π(var)= d ó drd ó drdrd, etc.
Π(i) = dr ó drrrdr ó drrrdrrrdr, etc.
Π(N) = r ó rr ó rrr ó rrrr, etc.

Ciertos trabajos teóricos permiten simplificar las expresiones de las DR-Cadenas procedentes de configuraciones iterativas manipulándolas como expresiones de caminos. Cuando se trata de una rutina, y no de un programa principal, las configuraciones primera y tercera se consideran como anormales, pero permiten identificar las variables de entrada y de salida respectivamente. En este caso, se puede verificar que las declaraciones del programador (en un lenguaje de alto nivel como Pascal por ejemplo) no contradicen el flujo tal y como se manifiesta a través de la ejecución.

Respecto a las instrucciones iterativas se puede añadir que existen fórmulas para representar las variables que aparecen en dichas estructuras:

Para las variables que aparezcan en la estructura while:
condicion (instrucciones condicion)*

para las que aparezcan en la estructura repeat:
(instrucciones condicion)+

y para las que aparezcan en la estructura for:

dada una sentencia for es más sencillo mostrarla por medio de una estructura while equivalente, tal y como ilustra el ejemplo:

```
for i:=1 to 5 do instrucciones;
```

seria equivalente a:

```
i:=1;
while i <=5 do begin
      instrucciones;
      i := i +1;
end;
```

de esta forma, la cadena correspondiente a la variable que controla la ejecución del for sería la formada por:

d r (instrucciones r d r) n

donde *condicion* es la cadena correspondiente a la variable en la condición de control de la estructura iterativa e *instrucciones* la cadena en la zona del conjunto de instrucciones que ejecuta la iteración.

2.3.2. Bucles infinitos

El análisis del flujo de datos es también aplicable al examen de las estructuras iterativas para detectar una cierta clase de bucles infinitos. Tomemos como ejemplo el algoritmo que evalúa la suma de los N primeros enteros:

```
Var i, j, numero, suma: integer;
Begin
      Read(numero);
      suma:= 0;
      i:= 1;
      While (i ≤ numero) do
      begin
            suma:= suma + 1;
            j:= i + 1;
      end;
      writeln('La suma es: ', suma);
End.
```

En el interior de la estructura *while* el programador ha cometido un error tipográfico y, en lugar de incrementar *i* con **i**:= i+1, ha escrito **j**:= i + 1. Es evidente que si el bucle *while* comienza a ejecutarse, no se terminará jamás. El análisis del flujo de datos es capaz de detectar este defecto. Y como vemos, en efecto, por lo menos una de las variables de control (las que aparecen ene el predicado de la estructura *while*) debe estar asignada (su DR-Cadena debe tener la forma ...d...) al menos una vez en el interior del bucle. Ahora bien, evaluando las DR-Cadenas de i y de Numero, obtenemos rr y 1 respectivamente, lo que nos permite detectar el defecto. Eso no significa por supuesto que todas las clases de bucles infinitos sean detectables. Sabemos que esta tarea es formalmente imposible. Es suficiente para convencerse suponer que el programador hubiera escrito :

i:= i – 1 en lugar de i:= i + 1

2.3.3. Parámetros

Tomemos un ejemplo concreto con el encabezamiento de un procedimiento, por ejemplo en Pascal:

```
Procedure ejemplo (var z:integer; x:integer);
     Const g = 5;
     Var a,b,c; integer;
```

Analizando este encabezamiento que expresa las intenciones del programador relativas a la naturaleza de uso de las diferentes variables que seguirán, podemos extraer la siguiente información:

- La variable "z" (que no es ni un fichero ni una tabla) se declara como una salida del algoritmo, es decir, como una variable que va a ver modificado su valor antes de salir del procedimiento, por ello su Dr-Cadena debe tener la forma: **...d...** En otras palabras, debe estar definida al menos una vez en el cuerpo del

procedimiento. ¿De qué serviría, en efecto, tener una rutina con z como resultado, si ésa nunca se definirá en su interior?

- La variable "x" es una entrada al procedimiento, es decir, no cambiará su valor, por ello su DR-Cadena debe ser del tipo **r...r**. En efecto, ¿de qué serviría una variable de entrada si este valor fuese inmediatamente modificado en el interior de la rutina? Continuando con el razonamiento, la DR-Cadena de "z" no debe terminarse con una definición, puesto que ésta última sería inútil (la rutina *proc* conoce únicamente el valor de "z" y no su dirección).

- En cuanto a "g", que es una constante local, su DR-Cadena debe estar formada únicamente por referencias.

- Finalmente, las variables "a", "b" y "c" están declaradas como auxiliares, lo que significa que sus DR-Cadenas no deben comenzar por una referencia (r) y tampoco acabar con una asignación (d), puesto que esta útima definición sería inútil, al estar limitado el alcance de estas variables en el interior de *proc*. Sus DR-Cadenas deben tomar la forma: **d..r**.

Estas DR-Cadenas se estudian de la misma forma que las del programa principal, es decir, como se ha visto en el ejemplo del programa Ejemplo.

Para entenderlo mejor supongamos ahora que un programador principiante ha codificado de la forma siguiente el interior de la rutina *proc*:

```
Begin
     a:= g*2 + z
     x:= z + c
end;
```

Se calculan las DR-Cadenas efectivas de las variables y comparan con la forma esperada, según las intenciones del programador definidas en el encabezamiento. Los resultados se resumen en la figura 2.2.

La comparación entre las DR-Cadenas efectivas y las que son "teóricas" (según las intenciones del programador) puede hacerse mediante una herramienta automática que detecte estas incoherencias. Se localizan así errores bastante comunes, sutiles y a veces peligrosos.

Variable	**DR-Cadena efectiva en el interior de proc**	**Forma teórica según las intenciones del programador declaradas en el encabezamiento**	**Conclusión**
z	r	...d...	¿Por qué z está declarado en var?
x	rd	r...r	¿Para qué sirve la asignación de x?
g	r	r	Utilización correcta
a	d	d…r	¿Para qué sirve la asignación de a?, y en general ¿Para qué sirve esta variable?
b	1	d…r	¿Por qué se declara b?
c	r	d…r	Peligroso: La variable no tiene valor inicial

Figura (2.2)

Como nota aclaratoria respecto a los parámetros, decir que todos los errores detectados en los parámetros de salida (excepto los producidos en el comienzo y

final de DR-Cadena), también formarán parte de los errores pertenecientes a la variable que se pasa como parámetro a la función. Esto se debe a que la DR-Cadena que se obtiene de la función, es la misma para las dos variables y aunque esto parezca repetitivo (aparecen dos veces el mismo tipo de error contenidos en las mismas líneas), no lo es, apuesto que se trata de dos variables diferentes que en algún momento toman los mismos valores en su DR-Cadena.

En cuanto a los parámetros pertenecientes a las funciones, surge una pregunta a la que ni expertos en compiladores han podido dar respuesta. ¿Qué DR-Cadena se produce, cuando a una función se le pasa como parámetros de entrada y de salida, la misma variable? Veamos el siguiente ejemplo:

```
Procedure ejemplo (Var x:integer, y:integer)
.......
.......
begin
      .......
      ejemplo(a,a);
      .......
end.
```

Aparece la llamada al procedimiento "ejemplo" con la misma variable (a), pasada primero como parámetro de salida y después como entrada. ¿Se introduce primero la referencia en la DR-Cadena de la variable o se analiza el comportamiento de la misma en la función (parámetro de salida) y una vez determinada la DR-Cadena, se introduce la referencia?

Realmente si se utiliza la lógica, tiene más sentido realizar primera una copia (parámetro de entrada) de la variable (para almacenar el valor al empezar la función) y posteriormente realizar los cambios pertinentes en la función, al pasar la dirección de memoria correspondiente (parámetro de salida). Si se hiciera al contrario no tendría sentido, puesto que al pasar la dirección de memoria se modificaría el valor de la variable y si al final se hiciera una copia, no se obtendría el

valor que se desea tener al principio de la función en el parámetros pasado como entrada, pero esto no está demostrado.

Las DR-Cadenas de los parámetros pertenecientes a las funciones, deben cumplir la siguiente estructura para ser correctas:

TIPO DE PARÁMETRO	CONDICIONES
Salida de rutina	.....d.....
Entrada a rutina	r.....r

Π (parámetros pasados por valor) debe ser de la forma **r...r**. En este caso sería inútil modificar esta variable nada más entrar en la rutina, y del mismo modo sería inútil que su DR-Cadena terminara con una modificación, ya que al salir de la rutina esta última modificación se perderá.

Π(parámetros pasados por variable) debe ser **...d...**, ya que se necesita que sea modificada al menos una vez en el interior de la rutina.

Al trabajar con un programa principal en el que existen diversas funciones y procedimientos se deberá seguir el siguiente proceso de análisis de las variables:

- Analizar primero independientemente cada procedimiento marcando los errores observados.

- Analizar el programa principal incluyendo las llamadas a procedimientos y funciones de tal forma que:

    - Para cada variable que se pase por valor se incluye una r en su cadena al hacerse la llamada al procedimiento.

- Para cada variable que se pase por referencia se incluye la cadena correspondiente a la variable en el procedimiento al hacerse la llamada a éste.

Se puede deducir por tanto que cuando se trata de una rutina, y no de un programa principal, las configuraciones primera y tercera no son consideradas como anormales para todos los casos, ya que pueden pertenecer a diversos tipos de variables contenidas en la rutina, como son variables pasadas por referencia, variables pasadas por valor, variables locales y variables globales, y en no todos estos casos se consideran configuraciones anómalas. Así, las anomalías primera y tercera, en el caso de las rutinas, permiten identificar las variables de entrada y de salida respectivamente.

2.3.4. Limitaciones

Se utiliza la misma filosofía del análisis del flujo de datos para detectar usos dudosos. Por ejemplo:

```
X:= funcion (a) + a
```

O incluso:

```
If (funcion(a) = a) and (a<10)
```

dónde *funcion* es una función que modifica el valor de a (sea por defecto de borde, sea porque manipula la dirección). En este caso el programador debe estar atento, porque puede que el valor de la variable a después de su paso a *funcion* "no sea el mismo" que el deseado para utilizar a continuación.

El análisis del flujo de datos es una técnica relativamente poco expandida en el mundo de la prueba. Sin embargo, tiene unas ventajas indiscutibles, sobre todo a causa de la universalidad de los errores que detecta. Se trata de una técnica poco costosa, fácilmente automatizable. Puede también intervenir tanto en la fase unitaria

(análisis de las anomalías clásicas) como en la fase de integración (análisis de las declaraciones de los diferentes parámetros de las rutinas) .

Su primer inconveniente reside en el hecho de que detecta a veces defectos inexistentes. Por ejemplo, si tenemos una parte de código:

```
a:= 8;
If (b>10) then
     a:= 7;
else
     b:= a;
writeln(a);
```

se obtiene:

Π(a) = ddr ó drr

Se detecta un error, pues la configuración …dd… es considerada como patológica. Ahora bien, no se puede decir que el conjunto de las instrucciones precedentes presenta una anomalía cualquiera: la variable "a" está inicializada, eventualmente asignada a "b", o modificada para ser finalmente imprimida. Una solución consistiría en no tomar en cuenta todas las DR-Cadenas irregulares que pertenezcan a expresiones con ó, como en el caso anterior. Pero esta solución es también inaceptable, pues no será capaz de detectar anomalías que son efectivas. Por ejemplo, en la configuración:

```
a:= 8;
If (b>10) then
     a:= 7;
else
     writeln(a);
a:= 0;
writeln(a);
```

se obtiene:

Π(a) = dddr ó drdr

Ahora bien, si según el razonamiento anterior, no se considera ddd, que da drdr que es normal. Es, por tanto, evidente que el conjunto de las instrucciones anteriores es irregular en la medida en que la asignación a:= 7 (después del *then*) es completamente inútil.

La segunda limitación del análisis del flujo de datos es más clásica, pues proviene del hecho de que es una técnica de análisis estático. Los punteros y las tablas no pueden ser tratados, debido a que es imposible saber cuál es la variable en la que tenemos la definición o la referencia, o saber a qué elemento de la tabla se referirá el índice durante la ejecución. Nos encontramos en este último caso enfrentados al mismo "dilema" anterior. Si la tabla es considerada como un solo único elemento, se señalará una anomalía inexistente:

```
a[i]:= 0;
a[i+1]:= 7;
```

con:

Π(a) = dd

Por otra parte, si se decide no detectar este tipo de anomalías sobre las tablas, nos arriesgamos a no detectar esquemas efectivamente irregulares. Por ejemplo:

```
a[1]:= 0;
a[1]:= 0;
```

Estos problemas del análisis del flujo de datos han conducido a ciertos investigadores a adoptar una aproximación **dinámica** del flujo de datos.

Para ello no se trata una tabla como una única variable, sino que cada una de las posiciones de la tabla es tratada como una variable. Se tratan de forma independiente y cuando en alguna línea del código aparece alguna de ellas, se modifica su DR-Cadena. De este modo esta técnica de análisis del flujo de datos que es estática, se convierte en dinámica para el estudio de estas estructuras.

2.3.5. Conclusiones

Finalmente decir, que el análisis del flujo de datos puede ser utilizado para comparar las diferentes transformaciones sufridas por una variable en un programa con las transformaciones teóricas y abstractas especificadas en los Diagramas de Flujo de Datos que han precedido a la fase de codificación.

3. OBJETIVOS

En el primer capítulo de este trabajo, se ha tratado de dar una introducción a las pruebas de programas. Seguidamente, en el segundo capítulo, se ha visto cuáles son las prácticas corrientes y las técnicas utilizadas desde una perspectiva de prueba del software, dando la clasificación de estas pruebas en base a diferentes criterios. Es el momento de introducirse más a fondo en la finalidad de este trabajo.

3.1. OBJETIVOS DEL PROYECTO

Los objetivos fundamentales que se pretenden cumplir en el desarrollo de este proyecto fin de carrera son los siguientes:

- Diseñar una aplicación que partiendo de un código fuente de prueba que está debidamente compilado (entendiéndose que no contiene errores semánticos), sea un poco más explícito en la prueba de programas y no limitarse única y exclusivamente a los errores esenciales ya vistos con anterioridad.

  Se busca que sea capaz de:

  - realizar un estudio de los errores que pueden producirse en una estructura (*if*, *while*, *repeat*, *for*...) o en varias de ellas consecutivas,
  - realizar un estudio de los errores que pueden encontrarse en una función, indistintamente de si la función se encuentra en el programa principal o dentro de otra función, ya sea localizando el error entre los parámetros (de entrada o de salida) o entre las variables locales declaradas en esta función,
  - realizar un control de las llamadas a funciones del sistema y llamadas a funciones definidas por el programador desde el programa principal,
  - almacenar cada aparición de cada variable durante el código, indistintamente de si es local, global, etc... tomando el número de línea en la que aparece así como también la posición que ocupa en esa línea

respecto a las demás variables para la posterior composición de su DRCadena,

- realizar un estudio de las cadenas generadas por las variables pasadas por referencia y por las variables globales en las funciones llamadas, para posteriormente añadirlas a las DrCadenas de las variables que corresponda(los parámetros),
- analizar y estudiar cada una de las DrCadenas generadas para poder identificar los posibles errores,
- tener la posibilidad de hacer la llamada a una función teniendo como parámetros una variable repetidas veces, pudiendo ser tanto por valor como referencia indistintamente. Por ejemplo, la siguiente llamada a la función prueba:

  Prueba(x,x,y,x,y,y)

  Esto se explicará más detalladamente en el capítulo 5.
- mostrar los errores por pantalla de forma clara y ordenada, y en un fichero de salida que genera la aplicación mostrar la descripción de la estructura de datos, indicando todas las funciones, variables y apariciones de esas variables que aparecen a lo largo del código.

- Diseñar una estructura basada en objetos y clases que pueda almacenar toda aquella información necesaria, extraída del código fuente, para implementar la técnica funcional de análisis del flujo de datos.

- Desarrollar esa aplicación bajo un lenguaje de programación orientado a objetos y así llevar a cabo la encapsulación y jerarquía de clases necesaria.

4. ENTORNO DE TRABAJO

En este capítulo se trata de introducir las herramientas utilizadas para la realización del proyecto, además de resumir brevemente las características más importantes que presentan éstas herramientas.

4.1. EVOLUCIÓN DE C

El desarrollo de C se remonta a la década de los setenta y se suele ligar al desarrollo del Unix, para el cual y en el cual fueron evolucionando las primeras versiones del C sobre PDP-11. Éste empezó como una evolución del lenguaje B, el lenguaje que se utilizó para las primeras versiones del Unix sobre PDP-7. También influyeron bastante otros lenguajes como el BCPL.

Inicialmente el C fue desarrollado por Dennis Ritchie en los laboratorios BELL sin ningún ánimo de ser revolucionario, novedoso o magnífico. Su único objetivo era tener un lenguaje práctico y flexible par el desarrollo del Unix. Poco a poco se vio que el lenguaje era útil para muchas más cosas que para lo que fue pensado. Citemos un párrafo del libro *El lenguaje de programación C*, de Brian W. Kernighan y Dennis M. Ritchie: *"El C es un lenguaje de programación de propósito general que ofrece economía de expresión, estructuras de flujo y control de flujo modernos y un conjunto rico de operadores. El C no es un lenguaje de muy alto nivel, tampoco un gran lenguaje, y no está especializado en ninguna área de aplicación. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que lenguajes supuestamente más poderosos."*

El C es un lenguaje que permite romper las reglas en el momento que sea necesario, permite ser más original en la confección de programas y supone que el programador sabe muy bien lo que está haciendo. Podríamos decir que el C es un lenguaje para programadores maduros que no necesitan que nadie les diga lo que pueden y no pueden hacer.

Una característica del C, es la ausencia de operaciones de alto nivel como manejo de cadenas, conjuntos, listas, vectores considerados como un todo, etc. El desarrollo de tales operaciones se dejan al programador para que éste las adapte a sus necesidades. De todas formas, siempre se pueden acudir a librerías estándar que implementen estas funciones.

Las mayores virtudes de C son:

- La aritmética de direcciones, su uso exhaustivo de punteros para casi cualquier cosa.
- Una eficiencia propia de lenguajes de muy bajo nivel.

Los mayores defectos de C:

- Una sintaxis demasiado peculiar por su excesiva economía.
- Falta de chequeo de tipos
- Falta de estandarización

Estos dos últimos inconvenientes se consiguieron solventar gracias a la aparición del estándar ANSI C. Un estándar que nació del esfuerzo del Instituto Nacional Americano para los Estándares (ANSI) y de la Organización Internacional de Estándares (ISO) y fue diseñado para "codificar lo existente en la práctica". El estándar ANSI C, añadió cosas nuevas, como los tipos enumerados, los prototipos de funciones y fija algunos aspectos que no estaban estandarizados o eran ambiguos. También fue muy importante la estandarización de las librerías.

4.2. LENGUAJE DE PROGRAMACIÓN C

Como ya se ha comentado en el apartado anterior, en un principio estaba inspirado en el lenguaje B. Después se creó el lenguaje C, que proporcionó la novedad del diseño de tipos y estructuras de datos.

Los tipos básicos de datos en un principio eran *char* (carácter), *int* (entero), *float* (reales en simple precisión) y *double* (reales en doble precisión). Posteriormente

se añadieron los tipos *short* (enteros de longitud menor o igual a la de un int), *long* (enteros de longitud mayor o igual a la de un int), *unsigned* (enteros sin signo) y *enumeraciones*. Los tipos estructurados básicos de C son las estructuras, las uniones y los arrays. Estos permiten la definición y declaración de tipos derivados de mayor complejidad.

Las instrucciones del lenguaje C son: *if*, *for*, *while*, *switch-case* e incluye punteros y funciones. Los argumentos de las funciones se pasan como entradas (no se modifican los valores) o como salidas (cuando se modifican los valores). Por otra parte, cualquier función puede ser llamada recursivamente.

El C posee bastante riqueza en sus operadores.

Hay toda una serie de operaciones que pueden hacerse con el lenguaje C, que realmente no están incluidas en el compilador propiamente dicho, sino que las realiza un preprocesador justo antes de cada compilación. Las dos más importantes son #define (directriz de sustitución simbólica o de definición) e #include (directriz de inclusión en el fichero fuente).

Finalmente C, que ha sido pensado para ser altamente transportable, igual que otros lenguajes tiene inconvenientes:

- La excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente no se detectan a simple vista.

- Por otra parte las precedencias de los operadores convierten a veces las expresiones en pequeños rompecabezas.

A pesar de todo, C ha demostrado ser un lenguaje eficaz y expresivo.

Este lenguaje ha evolucionado paralelamente a Unix, así en 1980, se añaden al lenguaje C, características como: clases, chequeo y conversión de los tipos de argumentos de función, etc. El resultado fue el lenguaje denominado "C con clases".

En 1983/84, "C con clases" fue rediseñado, extendido y nuevamente implementado. El resultado se denominó "Lenguaje C++". Las extensiones principales fueron funciones virtuales y operadores con varias funciones. Después de algún otro refinamiento más, C++ queda disponible en 1985. Este lenguaje fue inventado por Bjarne Strustrup (AT&T Bell Laboratories) y documentado por varios libros suyos.

4.3. EDITOR VISUAL C++

El "Visual C++" es un entorno de desarrollo y un conjunto de herramientas que permiten a los desarrolladores crear aplicaciones para Windows con un nivel de facilidad y rapidez que nunca se ha visto en otro entornos de C++. Sus herramientas han sido considerablemente mejoradas si se compara con otros editores del lenguaje de programación C++, como por ejemplo "Borland C++".Algunas de sus características son:

- Permite editar, compilar, ejecutar, depurar, recompilar, leer y escribir programas sin salir del entorno de programación de C++ y de una forma fácil y sencilla.

- Incluye muchas de las características de un depurador profesional, lo cual hace fácil la depuración de un programa.

- Ofrece una gran variedad de clases y plantillas ya diseñadas y precompiladas, que facilitan el manejo de estructuras así como también diversas tareas, como pueden ser interacción con el usuario y el manejo de excepciones.

- Ofrece varias estructuras como son grupos de funciones y macros que ayudan en la programación.

- Dispone de un panel (el panel "Output") donde se nos presenta cualquier tipo de información que deba proporcionarnos. Además, es donde podremos observar las instrucciones de progreso del compilador, las advertencias y los mensajes de error, y es el lugar en donde el depurador de Visual C++ despliega todas las variables con sus valores actuales de acuerdo a su avance a través del código.

- Dispone de un área (el panel "Workspace") que será la clave para navegar a través de los diversos componentes y partes de los proyectos en desarrollo, que nos mostrará las partes de nuestra aplicación de tres formas distintas: permitiéndonos navegar y manipular el código fuente en un nivel de clase C++, permitiéndonos encontrar y editar cada uno de los diferentes recursos de la aplicación, incluyendo los diseños, iconos y menús de la ventana de dialogo y por último nos permitirá observar y navegar a través de todos los archivos que conforman nuestra aplicación.

- Dispone de un área de edición, que es donde realizaremos todas nuestra edición cuando usemos "Visual C++" de un modo fácil y rápido; también es el lugar donde se desplegarán las ventanas necesarias para editar el código fuente de C++, así como el lugar donde se desplegará la ventana necesaria para diseñar un cuadro de diálogo. En el área de edición también tendremos el editor de iconos necesario para diseñar los iconos que utilicemos en nuestras aplicaciones.

- Además tenemos a nuestra disposición diversas barras de menús para trabajar con archivos, clases, herramientas estándar y comandos de ejecución y construcción de una forma más fácil e intuitiva.

- Dispondremos también de diversos tutoriales que nos harán más fácil tareas tales como el manejo de diálogos o el crear y manipular clases y variables.

- Dispone de ayuda acerca del editor, de cada uno de los paneles, de la sintaxis de las instrucciones, librerías, clases, tipos de excepciones y todo ello sin abandonar el editor.

4.4. LENGUAJE DE PROGRAMACIÓN C++

El C++ fue diseñado a mediados de los ochenta por el danés Bjarne Stroustrup en los laboratorios AT&T Bell, para facilitar la transición desde la programación tradicional, como el C, a estilos de programación basados en abstracción de datos y técnicas orientadas a objetos, que no han sido necesarias en este trabajo.

A primera vista, el C++ parece como un C extendido, aunque realmente sea mucho más que eso. Hereda la potencia de bajo nivel del C, la abstracción del Samalltalk y las capacidades de organización del Modula-2. Además, no hay nada que se pueda hacer en C que no se pueda hacer en C++. En sí mismo, el ANSI C es, en cierto modo, un subconjunto del C++.

En la actualidad, el C++ es un lenguaje versátil, potente y general. Su éxito entre los programadores profesionales le ha llevado a ocupar el primer puesto como herramienta de desarrollo de aplicaciones. El C++ mantiene las ventajas del C en cuanto a riqueza de operadores y expresiones, flexibilidad, concisión y eficiencia. Además, ha eliminado algunas de las dificultades y limitaciones del C original. La evolución de C++ ha continuado con la aparición de ***Java,*** un lenguaje creado simplificando algunas cosas de C++ y añadiendo otras, que se utiliza para realizar aplicaciones en Internet.

Las principales características de C++ son:

- Programación estructurada

- Programación orientada a objetos
- Economía en las expresiones
- Abundancia en operadores y tipos de datos
- Codificación en alto y bajo nivel simultáneamente
- Reemplaza ventajosamente la programación en ensamblador
- Utilización natural de las funciones primitivas del sistema
- No está orientado a ningún área en especial
- Producción de código objeto altamente optimizado
- Facilidad de aprendizaje

5. MÉTODO DE RESOLUCIÓN

En este capítulo se va a presentar el programa realizado para la prueba de programas de una forma más extensa.

Primeramente se presentará el manual técnico en el cual se mostrará la arquitectura de la aplicación explicando de forma amplia como trabaja el programa, las clases de las que se compone, como están implementadas esas clases, la funciones de las que esta compuesta la aplicación y se detallará que hace cada una de esas funciones.

A continuación se presentará el manual de usuario, que lo podemos considerar como una explicación acerca del manejo del programa.

5.1. MANUAL TÉCNICO

5.1.1. Funcionalidad del programa

El programa, en lo que respecta a su funcionalidad, lo podemos dividir en dos partes bien diferenciadas:

- por un lado tenemos la parte de lectura del código fuente, el cual se nos suministra mediante el fichero de entrada, y creación de la estructura de datos,

- y por otro tenemos la parte de análisis de las estructuras, y por tanto de las DrCadenas, y salida al usuario por pantalla y mediante el fichero de texto.

De hecho las funciones del programa las he dividido en dos bloques de acuerdo a la diferenciación anterior, para que sea más claro a que está dedicada cada función; un poco más adelante las explicaré más detenidamente.

Este programa de prueba de datos lee el código a analizar de forma lineal, es decir, lee desde la primera hasta la última línea de forma secuencial, no pega ningún tipo de salto durante la lectura puesto que no hace falta, ya que en el lenguaje Pascal es necesario definir todo lo que se va a usar (ya sean funciones o variables) en un momento muy específico, ya que en ese sentido no es un lenguaje muy libre.

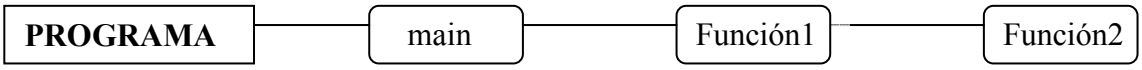
5.1.1.1. Lectura y creación de la estructura de datos

De forma global se puede explicar la primera parte del siguiente modo:

Paso 1: El programa comienza abriendo el fichero de entrada para su lectura y creando un objeto de la clase CPrograma en el cual podrá almacenar toda la información necesaria respecto a las funciones y a las variables del modo que se explica en la Figura 5.1. ; lo primero que hace la aplicación es leer el nombre del programa que estamos probando y crea el primer nodo de la lista de funciones, es decir un objeto de la clase CFuncion, que es el nodo de la función principal o programa principal y que llamaré 'main', por eso siempre el primer nodo de la lista de funciones es el nodo 'main' y se corresponde con el programa principal.

Paso 2: Después busca las variables que se declaran en el programa principal y se las añade a la lista dinámica de variables de ese nodo 'main', estas serán las variables locales y por eso su tipo será 'local'.

Paso 3: El siguiente paso es buscar las posibles funciones que haya definidas, esto se hace mediante un procedimiento que lo que hace es añadir nuevos nodos a la lista de funciones de 'Programa', solo que esta vez el nombre que se mete a estos nodos es el nombre de la función. Suponiendo que nuestro programa tuviese dos funciones la lista de las funciones quedaría como en el ejemplo 5.1.



Ejemplo 5.1.

Para cada función nueva que se añada a la lista se realizará un estudio de la misma, analizando y almacenando los parámetros como variables de la función (con tipo 'valor' o 'referencia') para luego seguir con un proceso similar a la lectura del programa principal, es decir, primero se leerán las variables, luego las funciones (en el caso de que existan) y por último el código. También se añadirán como variables globales de la función las variables locales de la función que esté en un nivel superior, y en el caso de que alguna de estas variables globales no sea usada durante el proceso de lectura de la función será eliminada al final del análisis de esta, esto se hace así porque a priori no sabemos que variables va a usar como globales una función, por eso le añadimos todas.

Paso 4: Después de la lectura de las funciones se procederá a la lectura del código, (mismo procedimiento que se usará para leer el código de las funciones) leyendo el código línea a línea y analizando cada una; estas líneas pueden contener:

- Sentencias – Para el análisis de las cuales existe una función específica que las separa para luego sacar y almacenar todas las variables que toman parte en ella, guardando en su lista de apariciones el valor ('R' o 'D').

- Estructuras – Existe una función para la identificación y lectura de cada tipo, además estas funciones son recursivas por si dentro de una estructura se llama a otra. Cuando se encuentra una estructura se almacena en la lista de apariciones de TODAS las variables que intervengan en esa función un símbolo que identifique el comienzo de la estructura y otro que identifique el final de la estructura, de este modo se controla perfectamente por donde ha pasado una variable

aunque nos podamos encontrar DrCadenas de tipo de 'D(o)()*' (para manejar estas cadenas se ha definido la función de limpieza de DrCadenas 'LimpiaDrCadena').

- Llamadas a funciones - Este es un tema más complejo puesto que para poder analizar con detenimiento cada llamada a una función es necesario crear una estructura mediante objetos tal y como se hace con las funciones y variables. Esta estructura se verá un poco mas adelante.

La lectura del código es un procedimiento iterativo que analiza línea a línea, sentencia a sentencia hasta que se identifica el final del bloque del código.

Paso 5: Una vez que se acaba la lectura del código se considera finalizada la estructura de funciones y variables, conformado por objetos de diferentas clases, tal y como se indica en la Figura 5.1. :

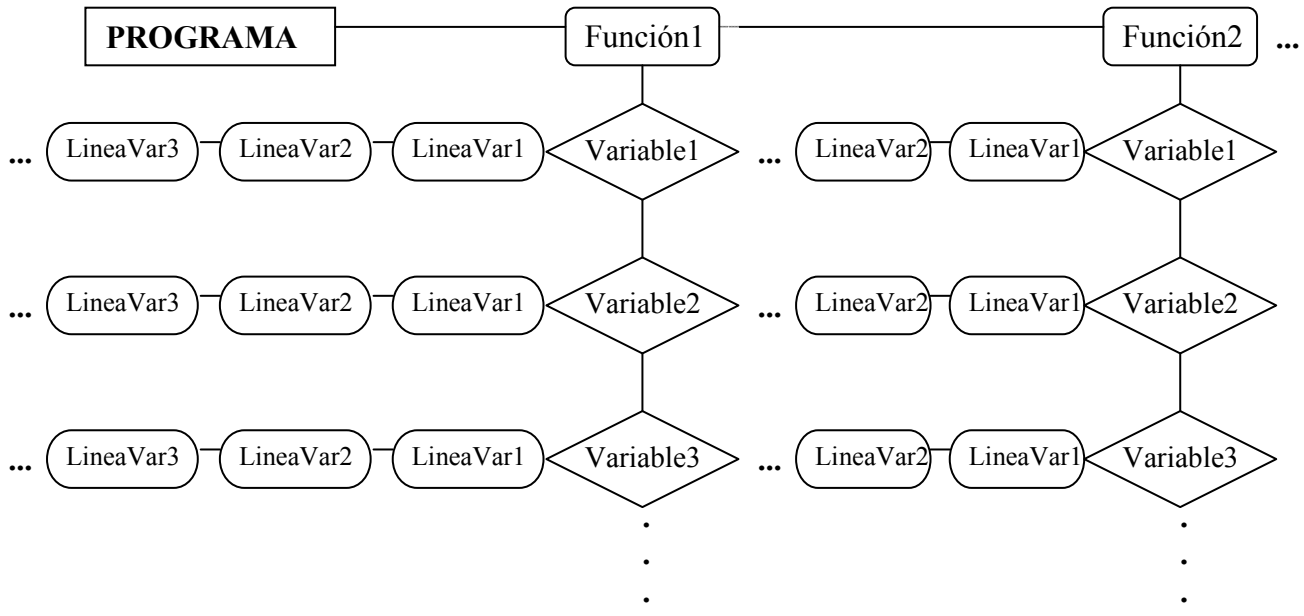


Figura 5.1.: Estructura de almacenamiento de las funciones

Siendo 'Función' objetos de la clase CVariable que guardan toda la información necesaria de las funciones y que forman una lista dinámica enlazada.

Siendo 'Variable' objetos de la clase CVariable que guardan toda la información necesaria de las variables y que forman una lista dinámica enlazada.

Siendo LineaVar la aparición de una variable en una línea de código ya sea siendo definida o referenciada. Guardan toda la información necesaria de las líneas en las que aparecen las variables y forman una lista dinámica enlazada.

Después inmediatamente se pasaría a la segunda parte del programa o parte análisis de las DrCadenas, pero antes de profundizar en esto explicaré como trata el programa las llamadas a funciones.

5.1.1.2. Llamadas a funciones

En esta parte se explicará más detenidamente el caso de que encontremos la llamada a una función, primero dando una justificación teórica y después explicando brevemente como resuelve este problema el programa y como usa las estructuras de datos.

5.1.1.2.1. Justificación teórica.

Los parámetros de una función pueden ser por valor (de entrada) o por referencia (de salida).

En los parámetros por valor se hace una copia de la dirección de memoria donde se encuentra almacenada la variable y por tanto a la DR-Cadena de la variable con la que se realiza la llamada se le añade una "r".

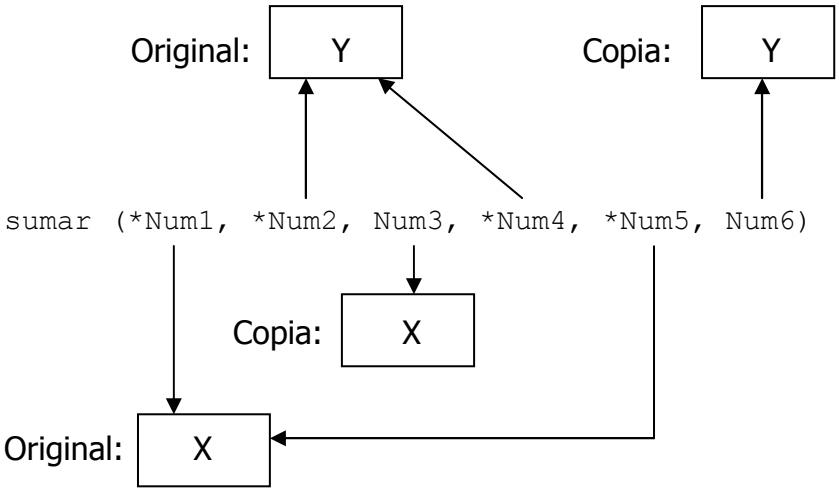
En los parámetros por referencia se genera un puntero que apunta a la dirección de memoria de la variable. Por lo tanto todas las modificaciones que se hagan en el parámetro se verán reflejadas en la DR-Cadena de la variable con la que se realiza la llamada.

En el programa de prueba puede ocurrir que nos encontremos ante una llamada a una función en la que se utilice una misma variable en varias posiciones. Por ejemplo si tenemos esta cabecera de función:

```
void sumar (int *Num1, int *Num2, int Num3, int *Num4, int *Num5, int Num6)
```

y la llamada

```
Sumar(x,y,x,y,x,y);
```



Para la variable x tendremos:

Posición 1: Por referencia

Posición 3: Por valor

Posición 5: Por referencia

La posición 1 y 5 apuntan a la posición de memoria donde se encuentra almacenada la variable X, por lo tanto todas las modificaciones y referencias que tengan la variable Num1 y Num5 se reflejaran en la variable X y la posición 3 realiza una copia del valor de la variable X, indicando únicamente una referencia. Su DR-Cadena será: primero una "r" por el parámetro pasado por valor (Num3) y después la DR-Cadena formada por los parámetros Num1 y Num5.

Y para la variable y tendremos:

Posición 2: Por referencia

Posición 4: Por referencia

Posición 6: Por valor

Las posiciones 2 y 4 apuntan a la posición de memoria donde se encuentra almacenada la variable Y, por lo que todas las modificaciones y referencias que tengan las variables Num2 y Num4 se reflejarán en la variable Y. Y la posición 6 realiza una copia del valor de la variable Y, indicando así una referencia a la variable Y. Su DR-Cadena será: primero una "r" por el parámetro pasado por valor (Num6) y después la DR-Cadena formada por los parámetros Num2 y Num4.

Como se puede observar no importa si el parámetro pasado por valor está antes que los pasados por referencia (hablando siempre de una misma variable), siempre primero se le asigna el valor del parámetro por valor (es decir, una "r") porque sino se hiciera así, sino en el orden en el que aparecen los parámetros, no tendría sentido ya que cuando una variable es pasada por valor es porque a ese parámetro se le quiere asignar el valor de la variable en la llamada. Si se hace según el orden en el que aparecen se le podría asignar un valor erróneo al parámetro por valor porque podría ser modificado por los valores asignados al haber sido pasado primero por referencia.

En cuanto a la utilización de variables globales en las llamadas a procedimientos, se puede decir que se tratan igual que si fueran una variable local de la función que realiza la llamada. Puede ocurrir que la variable global además de ser pasada como parámetro se encuentren referencias o asignaciones de ella en la función que se llama, en este caso su DR-Cadena sería una mezcla de la variable pasada y la variable global en sí (en el caso de que sea por referencia). Y para el caso en el que es por valor se le añadiría una "r" a la DR-Cadena y a continuación sus modificaciones en la función como variable global.

5.1.1.2.2. Tratamiento de las llamadas a funciones.

Primero de todo hay que indicar que cuando encontremos una llamada a una función el programa ya tendrá almacenado en su estructura de datos (Figura 5.1) toda la información necesaria acerca de esa función (siempre y cuando no estemos leyendo una función recursiva), puesto que en Pascal la definición de la función debe hacerse antes para poder llamar a esta durante el programa, y como la aplicación lee de forma lineal desde la primera línea hasta la última cuando lleguemos a la llamada de una función ya habrá pasado por la definición de la misma.

Cuando se alcanza la llamada a una función lo primero que se hace es crear un objeto de la clase CCabecera el cual almacenará todos los datos necesarios de los parámetros de la forma que ahora se indica:

La clase CCabecera se usa para almacenar los datos de la función y una lista dinámica enlazada de parámetros (CParametro), que son las variables pasadas en esta llamada a la función.

La clase CParametro se usa para almacenar los datos concernientes a un parámetro, incluyendo el nombre de la variable y una lista dinámica de posiciones (CPosicion) que indica que posición o posiciones toma cada parámetro en esa llamada a esa función.

La clase CPosicion almacena la posición que ocupa un determinado parámetro y tiene una lista dinámica de apariciones (CLineaVar) de esa variable a lo largo de la función llamada, es decir, la lista de apariciones de la variable que ocupa esa posición en la cabecera de definición de la función.

Los objetos de la clase CLineaVar son los mismos que los que se usan en la estructura de datos de las funciones y variables, es decir, muestran la aparición de una variable en una línea, ya sea cuando es referenciada o cuando es definida.

Esto es una forma resumida de la estructura y clases de los parámetros, en el apartado 5.1.2. se verá con más profundidad.

Tal y como he indicado cuando el programa encuentra una llamada a una función lee primero el nombre de la función llamada, después lee los parámetros almacenando también las posiciones y por ultimo busca esa función llamada en la estructura de funciones (Figura 5.1.) para acceder a la lista de apariciones de las variables que son parámetros por referencia en esa definición de función y así poder añadir a cada parámetro de la llamada la lista de apariciones (CLineaVar) que le corresponde; al final se crea una estructura de este tipo:

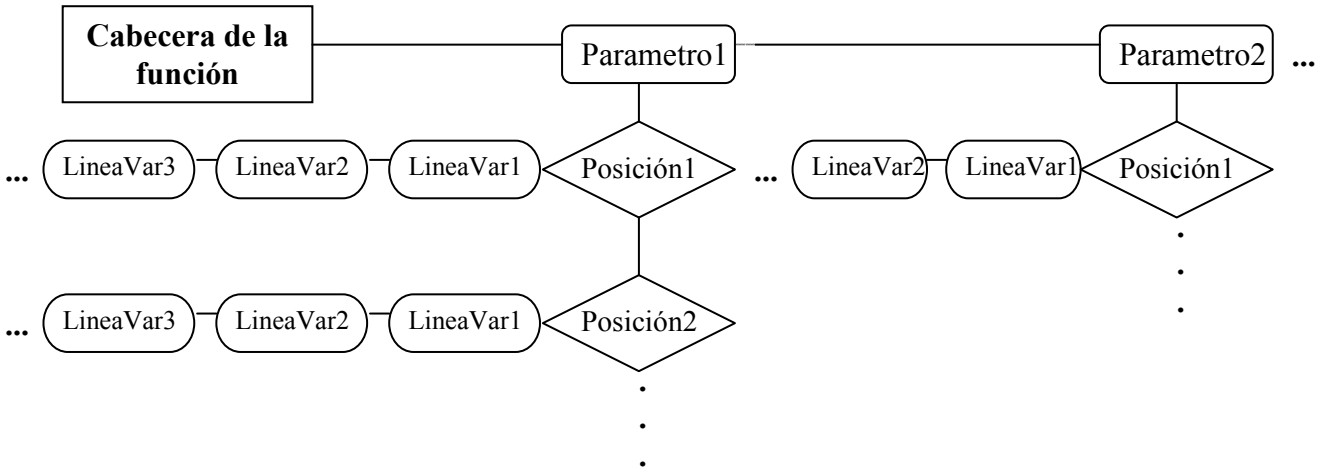


Figura 5.2.: Estructura de almacenamiento de las cabeceras

Que para mayor claridad voy a ilustrar mediante un ejemplo, ya que así es más fácil de comprender.

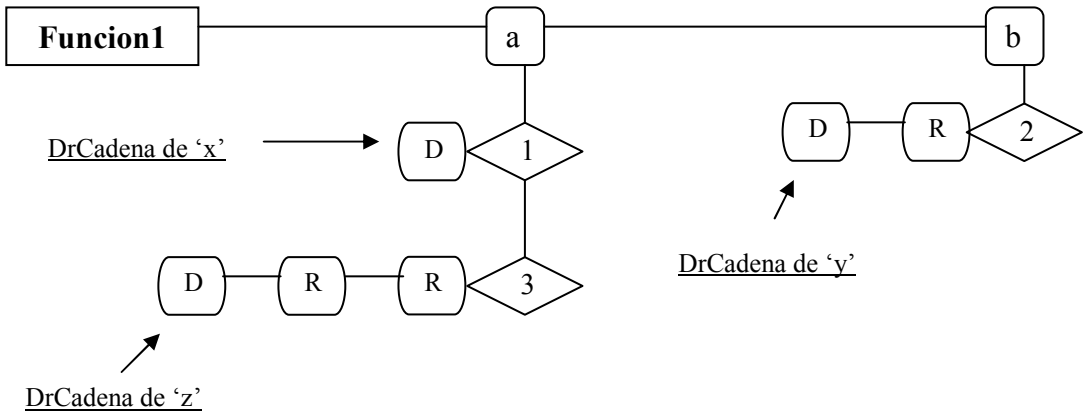
Si tenemos la siguiente función en Pascal:

```
Procedure Funcion1(Var x,y :int ; Var z :byte);
Begin
     x := y +1;
     y := z +1;
     z := z +1;
End;
```

Y la siguiente llamada a esa función:

```
Funcion1(a,b,a);
```

Nos daría como resultado la siguiente estructura:



Ejemplo 5.2.

De este modo, usando esta estructura para leer los parámetros englobamos también el caso de que una misma variable aparezca varias veces, ya que eso se soluciona mediante la clase CPosicion, según el ejemplo anterior la DrCadena que obtendría la variable 'a' sería la unión de las DrCadenas de las variables 'x' y 'z'.

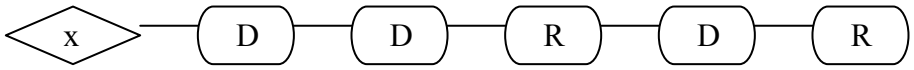
Las variables globales se tratan una vez se han leído las variables pasadas como parámetros y esta formada la estructura 'Cabecera'. Se comprueba si alguna de las variables que esta como parámetro además también esta como global, y en caso afirmativo se añade esa global como un parámetro más. Las globales que tenga la rutina y que no estén incluidas en la llamada a la función se tratan copiando su DrCadena como una aparición más de la variable, es decir, si una variable 'X' esta como global en una función 'F' y dentro de esa función 'X' tiene la DrCadena 'D', cada vez que se llame a 'F' a la lista de apariciones de 'X' se le añadirá 'D', y esto se hará tantas veces como se llame a 'F' a lo largo del programa.

5.1.1.3 Estudio de la estructura de datos y salida al usuario.

Para el estudio de las DrCadenas se presupone que se tiene ya totalmente acabada la lectura del código y por lo tanto tenemos la estructura de datos de las funciones integra (Figura5.1.).

Las DrCadenas de las variables se analizan una a una, es decir, variable a variable, y para sacar la DrCadena de cada variable lo que hace la aplicación es concatenar en orden la lista de apariciones que tenga la variable (CLineaVar), de este modo obtenemos la DrCadena tal y como la vamos a tratar. El proceso es tal y como es muestra en el siguiente ejemplo:

Si tenemos la variable 'x' con la lista de apariciones siguiente:



Ejemplo 5.3.

La DrCadena correspondiente de 'x' sería 'DDRDR'.

Una vez obtenida la DrCadena se pasa por un proceso de depurado el cual elimina elementos innecesarios de la cadena, tal como paréntesis vacíos o símbolos repetidos que se hacen innecesarios ('(R o R o R)').

Entonces es cuando comienza el análisis; este análisis va en función del tipo de variable que estemos tratando, ya que no es igual buscar los errores de un variable local que de una pasada por valor (Apartado 2.3.). En las variables locales se buscan errores tipo a), b) y c), mientras que en las pasadas como parámetro solo se busca el b), aparte de los suyos propios claro (r..#..r, #..d..#).

Ahora detallaré como se busca cada tipo de error, entendiendo el tipo a) como referencia inicial, el tipo b) como definiciones consecutivas y el tipo c) como definición final:

- Para buscar el error de la referencia inicial se crea una variable tipo char* que almacena, separándolos con espacios, todos los símbolos que pueden ser símbolos iniciales de la DrCadena y comprobando estos símbolos se saca que referencias pueden ser consideradas como iniciales y por tanto erróneas.

- Para buscar el error de la doble definición consecutiva se usa una función recursiva que para cada 'D' que exista en la cadena se le aplica un algoritmo el cual devuelve una variable tipo char* en la que están almacenados todos los símbolos que pueden ir inmediatamente después de esta 'D', ya sean 'D', 'R' o '1'. Se estudia esa variable con los símbolos y en el caso de que en la variable no exista ninguna 'R' y por el contrario si que exista alguna 'D', la 'D' estudiada es considerada como errónea ya que no puede ir seguida de 'R' alguna.

- Para buscar el error de la definición final se crea una variable tipo char* que almacena, separándolos con espacios, todos los símbolos que pueden ser símbolos finales de la DrCadena y comprobando estos símbolos se saca que definiciones pueden ser consideradas como finales y por tanto erróneas.

Este es el caso de las variables con tipo 'local', pero para aquellas que son 'referencia' o 'valor' es diferente pues tenemos que comprobar dos tipos de errores aparte:

- Las variables pasadas por valor deben tener una DrCadena de la forma R..#..R, no pueden tener una 'D' inicial puesto que entonces perderíamos el valor introducido y no nos serviría de nada, y tampoco pueden tener una 'D' final ya que al salir de la rutina esta definición se perdería. Para esto se

usan los mismos algoritmos que en los casos de los errores a) y c), tomando todos los símbolos iniciales y finales mediante variables tipo char*, y comprobando que no exista 'D' en ninguna de la variables, ya que en este caso esa 'D' seria errónea.

- Las variables pasadas por referencia tienen que tener una DrCadena de la forma #..D..#, ya que no tiene sentido que no sea modificada una variable de este tipo puesto que devuelve el valor. La forma de comprobar si existe el error es muy simple, tan solo se busca que haya alguna 'D' en la DrCadena, para ello se examinan todos los símbolos desde el principio hasta el fin, y en el caso de que no haya ninguna 'D' esta cadena se considerará errónea.

Una vez que se han identificado todos los diferentes errores las variables se muestran por pantalla una a una y en orden, mostrando para cada una su nombre, el nombre de la función en la que esta definida, su tipo y, en el caso de que los tenga, el tipo de error que hay; mostrando para cada error el símbolo concreto que lo produce y la línea de código en la cual esta.

Por último genera el fichero de texto, en el cual se escribe toda la estructura de las funciones y variables (Figura 5.1.) de forma ordenada para su posterior tratamiento mediante otras aplicaciones.

5.1.2. Estructuras del programa

Para el desarrollo del programa se han utilizado diferentes estructuras para el almacenamiento de los valores de las DrCadenas y para el almacenamiento de los parámetros.

5.1.2.1 Clases de las DrCadenas

La forma de almacenar los valores de las DrCadenas es mediante un grafo o una lista de listas. Para el diseño de la estructura de datos se ha considerado como que el programa se compone de funciones, la función de variables y cada variable de líneas, que son las líneas en las que aparece esa variable, tal y como indicó anteriormente (Figura 5.1).

Primero se crea un objeto de la clase CPrograma que guarda el nombre del programa a estudiar y contiene una lista enlazada de objetos de la clase CFuncion para poder controlar todas las funciones que aparezcan a lo largo del programa.

Cada objeto de la clase CFuncion que compone la lista almacena el nombre de la función, la línea en la que comienza el código de la función, y la línea en la que termina el código de la función, además contiene una lista enlazada de objetos de la clase CVariable para poder controlar todas las variables que aparezcan a lo largo de la función.

Cada objeto de la clase CVariable que compone la lista almacena el nombre de la variable, el tipo ('local', 'valor', 'referencia' o 'global'), y la posición que ocupa en la cabecera en el caso de que sea por valor o por referencia, además contiene una lista enlazada de objetos de la clase CLineaVar para poder conocer todas las líneas en las que aparece esa variable.

Cada objeto de la clase CLineaVar que compone la lista almacena el número de línea en la que aparece la variable, la posición en la que está la variable dentro de la cadena, y el valor de esa aparición, es decir, si es 'R', 'D', '1', el comienzo de una estructura o el final de una estructura.

Aunque en la Figura 5.1. se mostró esquemáticamente la estructura de datos utilizada, a continuación se presentará de una modo más formal el diseño se la estructura que se ha usado mediante clases y objetos; para ello me basaré en el "Lenguaje de Modelado Unificado" (UML):

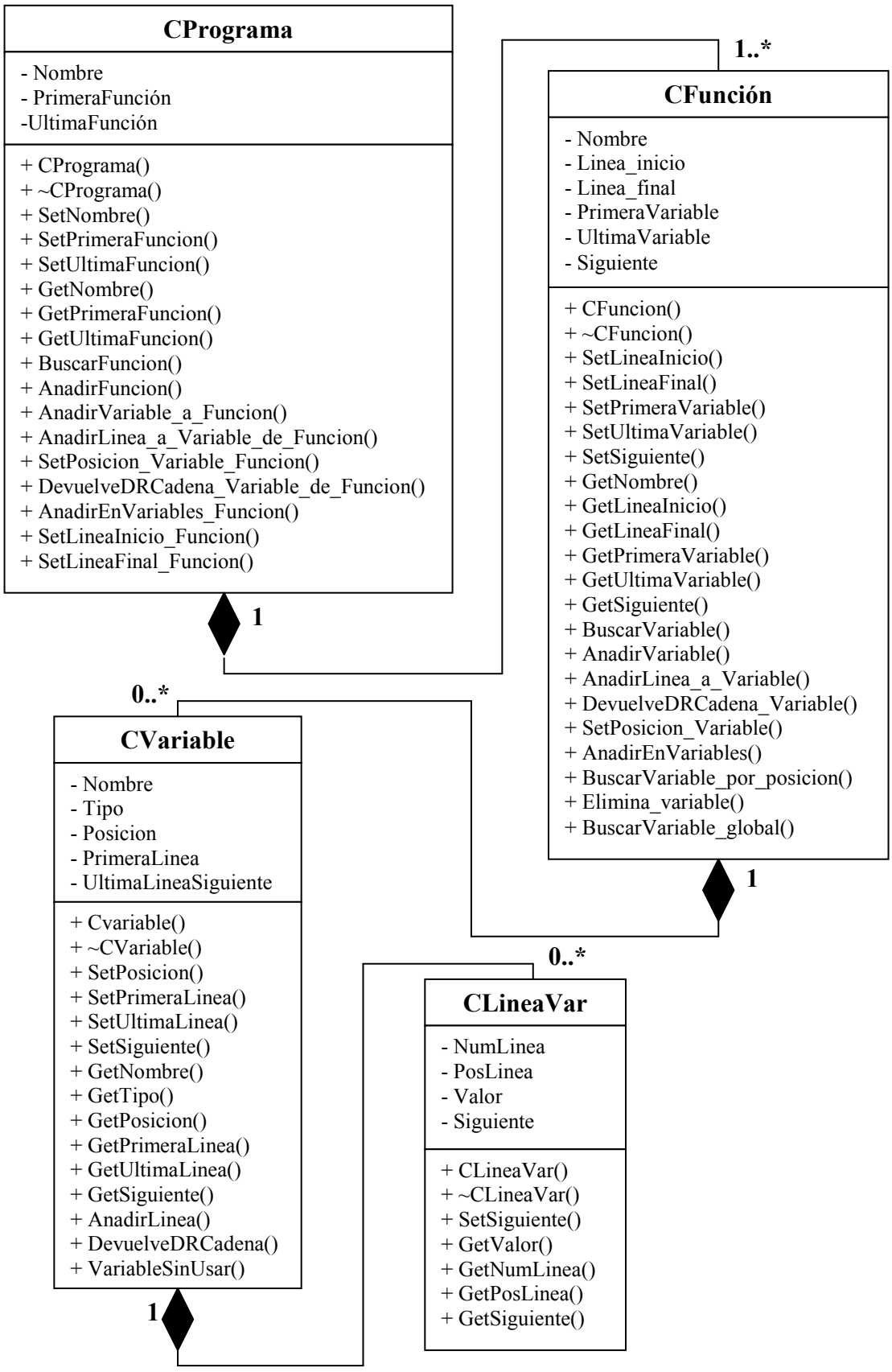


Figura 5.3: Diagrama de clases UML de las clases de las funciones

A continuación se mostrará y explicará cada una de las clases más detenidamente resumiendo brevemente cada una de sus funciones.

La clase CPrograma

- Atributos:
  - Nombre: Variable tipo *char** que almacena el nombre del programa.
  - PrimeraFuncion: Variable tipo CFuncion*, que apunta al primer elemento de la lista de funciones.
  - UltimaFuncion: Variable tipo CFuncion*, que apunta al último elemento de la lista de funciones.
- Funciones:
  - CPrograma: Constructor de la clase.
  - ~CPrograma: Destructor de la clase.
  - SetNombre: Función para dar valor al atributo 'Nombre'.
  - SetPrimeraFuncion: Función para dar valor al atributo 'PrimeraFuncion'.
  - SetUltimaFuncion: Función para dar valor al atributo 'UltimaFuncion'.
  - GetNombre: Función para obtener el valor del atributo 'Nombre'.
  - GetPrimeraFuncion: Función para obtener el valor del atributo 'PrimeraFuncion'.
  - GetUltimaFuncion: Función para obtener el valor del atributo 'UltimaFuncion'.
  - BuscarFuncion: Función para buscar un nodo concreto de la lista de objetos de la clase CFuncion.
  - AnadirFuncion: Función que añade un nodo nuevo a la lista de objetos de la clase CFuncion.

- AnadirVariable_a_Funcion: Función que añade un nodo variable a la lista de objetos de la clase CVariable que tiene un objeto concreto de la clase CFuncion.
- AnadirLinea_a_Variable_de_Funcion: Función que añade un nodo nuevo a la lista de objetos de la clase CLineaVar que tiene un objeto concreto de la clase CVariable, que a su vez tiene un objeto concreto de la clase CFuncion, que esta en la lista de objetos CFuncion.
- SetPosicion_Variable_Funcion: Función para dar valor al atributo 'Posicion' de un nodo CVariable de la lista de objetos CVariable, que a su vez esta en un nodo CFuncion de la lista de objetos CFuncion.
- DevuelveDRCadena_Variable_de_Funcion: Función que devuelve la DrCadena de una variable concreta de una función concreta.
- AnadirEnVariables_Funcion: Función que añade un nuevo nodo CLineaVar a todos los objetos de la lista de variables de una determinada función.
- SetLineaInicio_Funcion: Función para dar valor al atributo 'Linea_Inicio' de una determinada función.
- SetLineaFinal_Funcion: Función para dar valor al atributo 'Linea_Final' de una determinada función.

La clase CFuncion

- Atributos:
  - Nombre: Variable tipo *char** que almacena el nombre de la función.
  - Linea_inicio: Variable tipo *int* que almacena la línea de código en la que empieza la función.
  - Linea_final: Variable tipo *int* que almacena la línea de código en la que finaliza la función

- PrimeraVariable: Variable tipo CVariable*, que apunta al primer elemento de la lista de variables.
- UltimaVariable: Variable tipo CVariable*, que apunta al último elemento de la lista de variables.
- Siguiente: Variable tipo CFuncion*, que apunta al siguiente elemento de la lista de funciones.

- Funciones:
  - CFuncion: Constructor de la clase.
  - ~CFuncion: Destructor de la clase.
  - SetLineaInicio: Función para dar valor al atributo 'Linea_inicio'.
  - SetLineaFinal: Función para dar valor al atributo 'Linea_final'.
  - SetPrimeraVariable: Función para dar valor al atributo 'PrimeraVariable'
  - SetUltimaFuncion: Función para dar valor al atributo 'UltimaVariable'.
  - SetSiguiente: Función para dar valor al atributo 'Siguiente'.
  - GetNombre: Función para obtener el valor del atributo 'Nombre'.
  - GetLineaInicio: Función para obtener el valor del atributo 'Linea_inicio'.
  - GetLineaFinal: Función para obtener el valor del atributo 'Linea_final'.
  - GetPrimeraVariable: Función para obtener el valor del atributo 'PrimeraVariable'.
  - GetUltimaVariable: Función para obtener el valor del atributo 'UltimaVariable'.
  - GetSiguiente: Función para obtener el valor del atributo 'Siguiente'.
  - BuscarVariable: Función para buscar un nodo concreto de la lista de objetos de la clase CVariable.

- AnadirVariable: Función que añade un nodo nuevo a la lista de objetos de la clase CVariable.
- AnadirLinea_a_Variable: Función que añade un nodo nuevo a la lista de objetos de la clase CLineaVar que tiene un objeto concreto de la clase CVariable.
- DevuelveDRCadena_Variable: Función que devuelve la DrCadena de una variable concreta de la lista de objetos CVariable.
- SetPosicion_Variable: Función para dar valor al atributo 'Posicion' de una variable concreta de la lista de objetos CVariable.
- AnadirEnVariables: Función que añade un nuevo nodo CLineaVar a todos los objetos de la lista de variables.
- BuscarVariable_por_posicion: Función que busca en la lista de objetos CVariable usando como índice de busca el atributo 'Posicion'.
- EliminaVariable: Función que elimina un nodo concreto de la lista de variables.
- BuscarVariable_global: Función que busca un nodo concreto de la lista de variables, teniendo en cuenta que su tipo debe ser 'global'.

La clase CVariable

- Atributos:
  - Nombre: Variable tipo *char** que almacena el nombre de la variable.
  - Tipo: Variable tipo *char** que almacena el tipo de la variable.
  - Posicion: Variable tipo *int* que almacena la posición de la variable dentro de la línea en la que esté.
  - PrimeraLinea: Variable tipo CLineaVar*, que apunta al primer elemento de la lista de apariciones.

- UltimaLinea: Variable tipo CLineaVar*, que apunta al último elemento de la lista de apariciones.
- Siguiente: Variable tipo CVariable*, que apunta al siguiente elemento de la lista de variables.

- Funciones:
  - CVariable: Constructor de la clase.
  - ~CVariable: Destructor de la clase.
  - SetPosicion: Función para dar valor al atributo 'Posicion'.
  - SetPrimeraLinea: Función para dar valor al atributo 'PrimeraLinea'
  - SetUltimaLinea: Función para dar valor al atributo 'UltimaLinea'.
  - SetSiguiente: Función para dar valor al atributo 'Siguiente'.
  - GetNombre: Función para obtener el valor del atributo 'Nombre'.
  - GetTipo: Función para obtener el valor del atributo 'Tipo'.
  - GetPosicion: Función para obtener el valor del atributo 'Posicion'.
  - GetPrimeraLinea: Función para obtener el valor del atributo 'PrimeraLinea'.
  - GetUltimaLinea: Función para obtener el valor del atributo 'UltimaLinea'.
  - GetSiguiente: Función para obtener el valor del atributo 'Siguiente'.
  - AnadirLinea: Función para añadir un nodo a la lista de objetos 'CLineaVar'.
  - DevuelveDRCadena: Función que concatena los valores de todos los nodos de la lista de apariciones y devuelve la DrCadena de la variable.
  - VariableSinUsar: Función que comprueba si la variable tiene lista de objetos 'CLineaVar' o no.

La clase CLineaVar

- Atributos:
  - NumLinea: Variable tipo *int* que almacena el número de línea en la que aparece la variable.
  - PosLinea: Variable tipo *int* que almacena en que posición dentro de la línea aparece la variable.
  - Valor: Variable tipo *char** que almacena el valor de la variable en esa línea ('D', 'R', '1', comienzo de estructura o fin de estructura).
  - Siguiente: Variable tipo CLineaVar*, que apunta al siguiente elemento de la lista de variables.
- Funciones:
  - CLineaVar: Constructor de la clase.
  - ~CLineaVar: Destructor de la clase.
  - SetSiguiente: Función para dar valor al atributo 'Siguiente'.
  - GetValor: Función para obtener el valor del atributo 'Valor'.
  - GetNumLinea: Función para obtener el valor del atributo 'NumLinea'.
  - GetPosLinea: Función para obtener el valor del atributo 'PosLinea'.
  - GetSiguiente: Función para obtener el valor del atributo 'Siguiente'.

5.1.2.2 Clases de los parámetros

La forma de almacenar los valores de los parámetros también es mediante un grafo o una lista de listas. Para el diseño de la estructura de datos se ha considerado como que una llamada a una función se compone de parámetros, cada parámetro de posiciones dentro de la cabecera y cada posición de apariciones, que son las líneas en las que aparece la variable que esta en la definición de la función en esa posición

(siempre y cuando esté pasada por referencia), tal y como indicó anteriormente (Figura 5.2).

Primero se crea un objeto de la clase CCabecera que guarda el nombre de la función llamada y contiene una lista enlazada de objetos de la clase CParametro para poder controlar todos los parámetros que aparezcan en la llamada a la función.

Cada objeto de la clase CParametro que compone la lista almacena el nombre de la variable pasada como parámetro y contiene una lista enlazada de objetos de la clase CPosicion para poder controlar en que posiciones aparece ese parámetro en la llamada a la función.

Cada objeto de la clase CPosicion que compone la lista almacena una posición que ocupa la variable en la cabecera de la llamada de la función, además contiene una lista enlazada de objetos de la clase CLineaVar para poder conocer todas las líneas en las que aparece la variable correspondiente (la de la cabecera de la definición).

Cada objeto de la clase CLineaVar que compone la lista almacena el número de línea en la que aparece la variable, la posición en la que está la variable dentro de la cadena, y el valor de esa aparición, es decir, si es 'R', 'D', '1', el comienzo de una estructura o el final de una estructura.

Aunque en la Figura 5.2. se mostró esquemáticamente la estructura de datos utilizada, a continuación se presentará de una modo más formal el diseño se la estructura que se ha usado mediante clases y objetos; para ello me basaré en el "Lenguaje de Modelado Unificado" (UML):

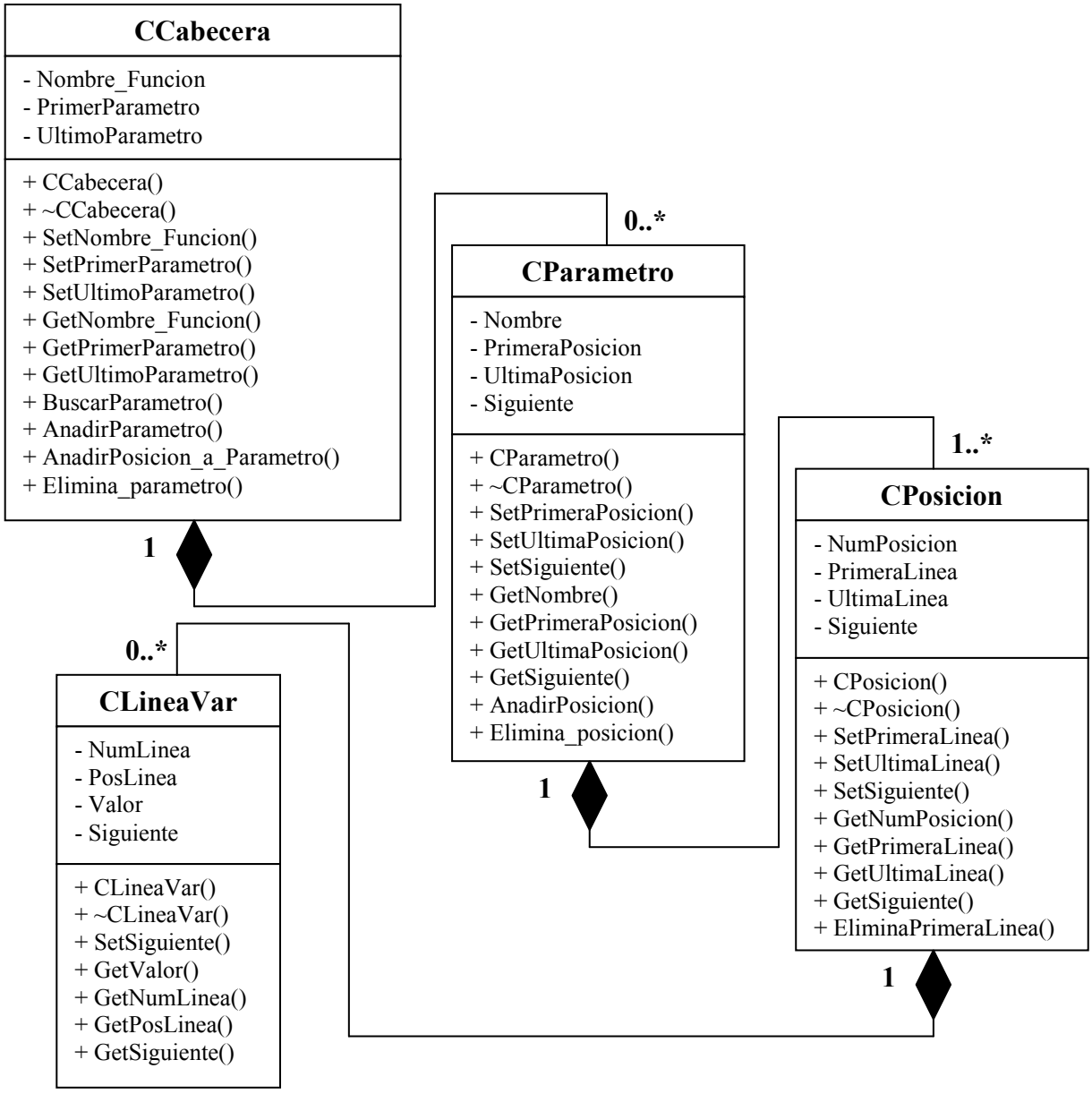


Figura 5.4: Diagrama de clases UML de las clases de los parámetros

A continuación se mostrará y explicará cada una de las clases más detenidamente resumiendo brevemente cada una de sus funciones.

La clase CCabecera

- Atributos:
  - Nombre_Funcion: Variable tipo *char** que almacena el nombre de la función llamada.

  - PrimerParametro: Variable tipo CParametro*, que apunta al primer elemento de la lista de parámetros.
  - UltimoParametro: Variable tipo CParametro*, que apunta al último elemento de la lista de parámetros.
- Funciones:
  - CCabecera: Constructor de la clase.
  - ~CCabecera: Destructor de la clase.
  - SetNombre_Funcion: Función para dar valor al atributo 'Nombre_Funcion'.
  - SetPrimerParametro: Función para dar valor al atributo 'PrimerParametro'.
  - SetUltimoParametro: Función para dar valor al atributo 'UltimoParametro'.
  - GetNombre_Funcion: Función para obtener el valor del atributo 'Nombre_Funcion'.
  - GetPrimerParametro: Función para obtener el valor del atributo 'PrimerParametro'.
  - GetUltimoParametro: Función para obtener el valor del atributo 'UltimoParametro'.
  - BuscarParametro: Función que busca en la lista de parámetros un determinado nodo.
  - AnadirParametro: Función que añade a la lista de parámetro un nuevo nodo.
  - AnadirPosicion_a_Parametro: Función que añade un nuevo nodo a la lista de objetos CPosicion que a su vez tiene un determinado nodo de la lista de parámetros.
  - Elimina_parametro: Función que elimina un determinado nodo de la lista de objetos CParametro.

La clase CParametro

- Atributos:

  - Nombre: Variable tipo *char** que almacena el nombre del parámetro.
  - PrimeraPosicion: Variable tipo CPosicion*, que apunta al primer elemento de la lista de posiciones.
  - UltimaPosicion: Variable tipo CPosicion*, que apunta al último elemento de la lista de posiciones.
  - Siguiente: Variable tipo CParametro*, que apunta al siguiente elemento de la lista de parámetros.
- Funciones:
  - CParametro: Constructor de la clase.
  - ~CParametro: Destructor de la clase.
  - SetPrimeraPosicion: Función para dar valor al atributo 'PrimeraPosicion'
  - SetUltimaPosicion: Función para dar valor al atributo 'UltimaPosicion'.
  - SetSiguiente: Función para dar valor al atributo 'Siguiente'.
  - GetNombre: Función para obtener el valor del atributo 'Nombre'.
  - GetPrimeraPosicion: Función para obtener el valor del atributo 'PrimeraPosicion'.
  - GetUltimaPosicion: Función para obtener el valor del atributo 'UltimaPosicion'.
  - GetSiguiente: Función para obtener el valor del atributo 'Siguiente'.
  - AnadirPosicion: Función para añadir un nodo a la lista de objetos CPosicion.
  - Elimina_posicion: Función para eliminar un nodo a la lista de objetos CPosicion.

La clase CPosicion

- Atributos:

- NumPosicion: Variable tipo *int* que almacena la posición del parámetro en la cabecera de la función llamada.
- PrimeraLinea: Variable tipo CLineaVar*, que apunta al primer elemento de la lista de apariciones.
- UltimaLinea: Variable tipo CLineaVar*, que apunta al último elemento de la lista de apariciones.
- Siguiente: Variable tipo CPosicion*, que apunta al siguiente elemento de la lista de posiciones.

- Funciones:
  - CPosicion: Constructor de la clase.
  - ~CPosicion: Destructor de la clase.
  - SetPrimeraLinea: Función para dar valor al atributo 'PrimeraLinea'
  - SetUltimaLinea: Función para dar valor al atributo 'UltimaLinea'.
  - SetSiguiente: Función para dar valor al atributo 'Siguiente'.
  - GetNumPosicion: Función para obtener el valor del atributo 'NumPosicion'.
  - GetPrimeraLinea: Función para obtener el valor del atributo 'PrimeraLinea'.
  - GetUltimaLinea: Función para obtener el valor del atributo 'UltimaLinea'.
  - GetSiguiente: Función para obtener el valor del atributo 'Siguiente'.
  - EliminaPrimeraLinea: Función para eliminar el primer nodo de la lista de objetos de CLineaVar.

La clase CLineaVar es exactamente la misma que se usa en la estructura de las DrCadenas, por lo tanto ya ha sido explicada en el apartado 5.1.2.1.

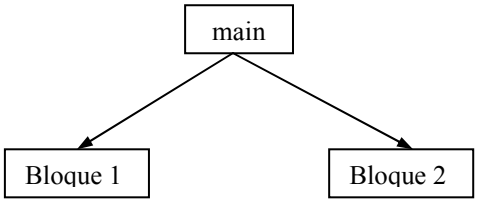
5.1.3. Funciones del programa

Las funciones de esta aplicación tal y como se dijo anteriormente se pueden dividir en dos bloques bien diferenciados:

- ✓ **Bloque 1:** Funciones que leen código, analizan código y crean la estructura de datos.

- ✓ **Bloque 2:** Funciones que tratan las DrCadenas, analizan las DrCadenas para encontrar errores y generan la salida al usuario.

Estos son los que podríamos llamar bloques principales, pero ambos bloques son llamados desde la función 'main', por lo que el esquema que nos quedaría sería:



Ejemplo 5.4

Este esquema es demasiado simple y general, para hacerlo de un modo más ilustrativo y específico voy a dividir las funciones en una mayor variedad de tipos.

Podemos englobar las funciones del Bloque1 en los siguiente tipos:

- Identificar estructuras.
- Leer bloques de código.
- Leer estructuras.
- Leer sentencias.
- Leer condiciones.
- Leer funciones.
- Leer definición de variables.
- Leer llamadas a funciones.

- Auxiliares.

Las funciones auxiliares son muy sencillas y son usadas por todo tipo de funciones por lo que las voy a suprimir del esquema. Las funciones de creación de la estructura son las funciones públicas de las clases vistas antes y también son usadas por todas estas funciones, pero no por otras de otro tipo, es por eso que tampoco las incluiré en el esquema. El esquema de las funciones del Bloque1 quedaría como se indica en la figura 5.4.

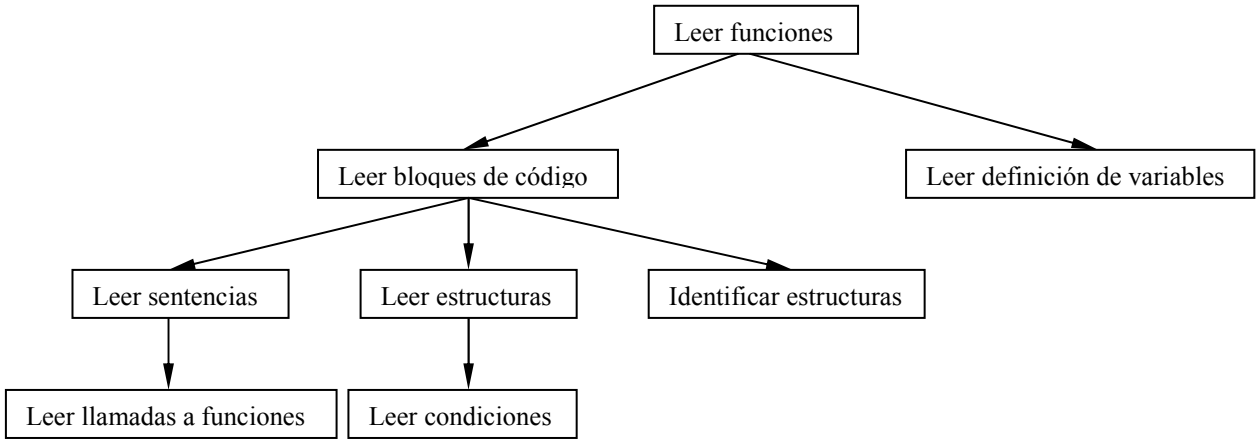


Figura 5.4: Esquema de funciones de lectura y análisis del código.

Las funciones del Bloque2 o funciones de análisis de las DrCadenas las podríamos clasificar del siguiente modo:

- Buscar elementos.
- Auxiliares en la búsqueda de elementos.
- Comprobar datos.
- Mostrar errores.

Las funciones auxiliares en la búsqueda de elementos son necesarias sobre todo para cuando haya iteraciones en la DrCadena puesto que, por ejemplo, para buscar los símbolos que siguen a otro son necesarios más algoritmos si nos encontramos con un *'while'* o un *'repeat'*. Las funciones necesarias para obtener información de la estructura de datos son las propias funciones públicas de las clases, por lo que no las mostraré en el esquema. Todas estas funciones son

llamadas desde la propia función de análisis. El esquema de las funciones del Bloque2 quedaría como se indica en la figura 5.5.

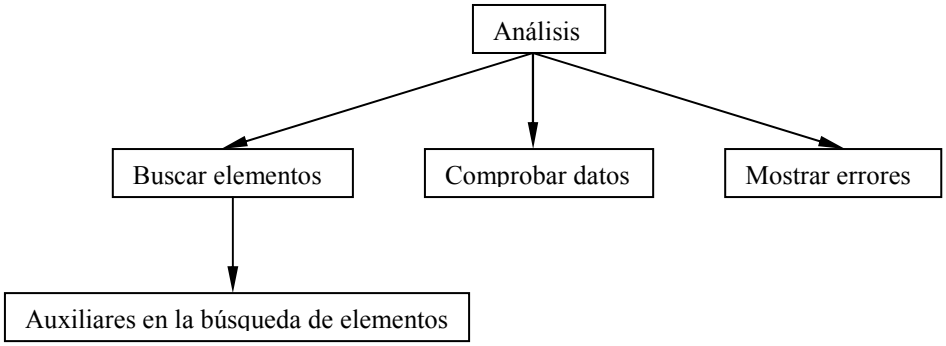


Figura 5.5: Esquema de funciones de análisis de DrCadenas y salida.

5.1.3.1 Funciones de lectura

Estas son las funciones de la aplicación que se encargan de tareas tales como identificar variables en su definición, identificar variables en las sentencias y condiciones, controlar la posición de cada variable que aparezca en una línea, identificar en una línea una estructura, identificar el tipo de estructura, leer y analizar cada estructura, identificar las definiciones de funciones, leer y analizar completamente una función, identificar el comienzo de un bloque de código, leer línea a línea cada bloque de código, leer y estudiar cada llamada a una función, introducir cada función en la estructura de datos, introducir cada variable en la estructura de datos, introducir cada aparición de cada variable en la estructura de datos y otras tareas consideradas menores.

En la figura 5.6. se muestra las llamadas entre las funciones más importantes englobadas dentro del Bloque1 mediante las cuales se lee y trata las líneas del código del programa de prueba introducido y se crea la estructura de datos.

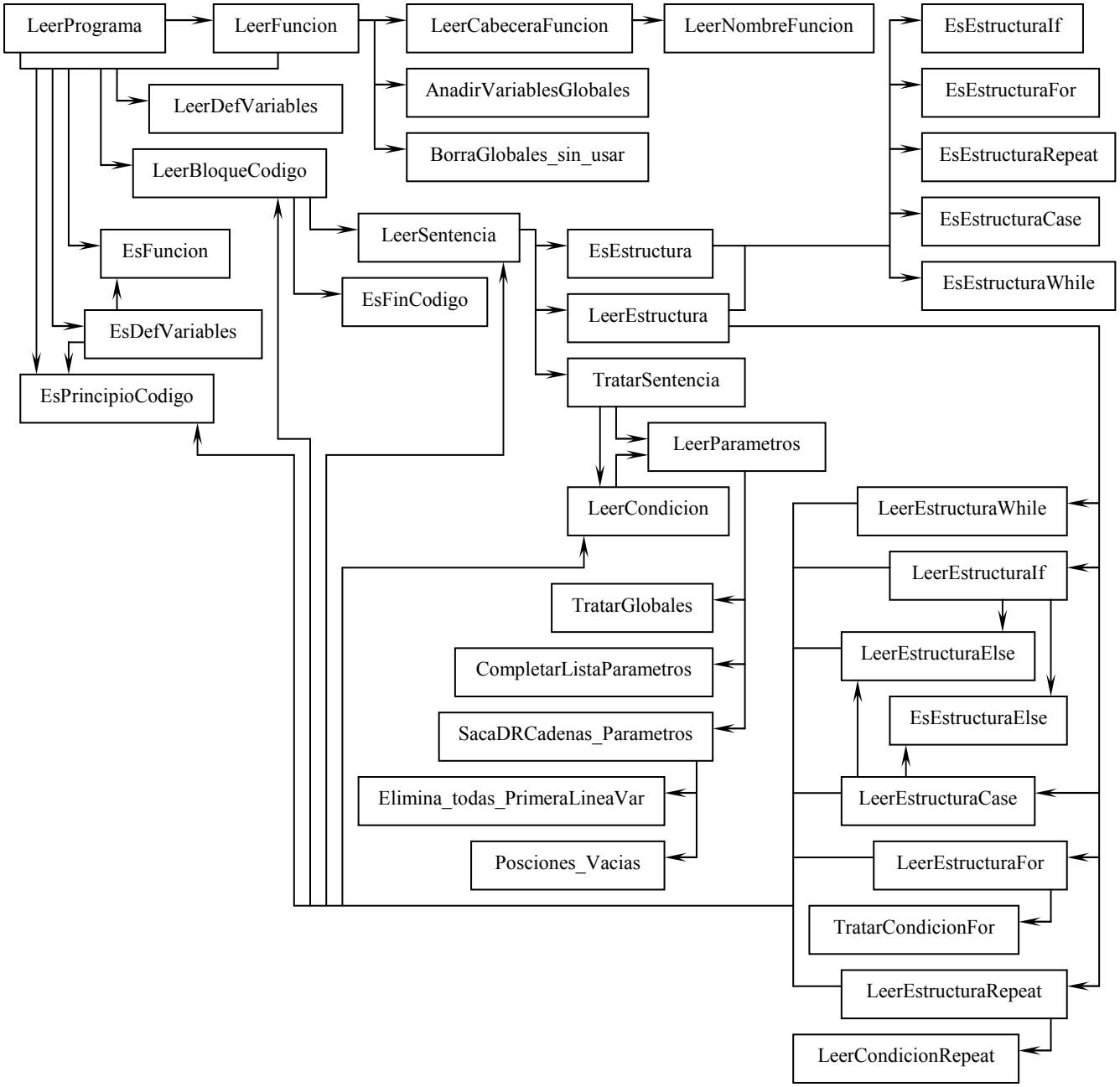


Figura 5.6: Esquema de llamadas de funciones de lectura y análisis del código.

A continuación se explicarán una a una y más detenidamente las funciones que englobé en el primer bloque o también llamadas de lectura y estudio del código.

La función EsFuncion

- Entradas:
  - Línea: Variable tipo char* de la que se desea saber si contiene la cabecera de la definición de una función.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra la cabecera de la definición de una función o no.

Funcionamiento

El funcionamiento de esta función es muy simple, tan solo busca en la línea pasada por valor la palabra "procedure" o "function" , que identificarían esa línea inequívocamente como la cabecera de definición de una función. En caso positivo devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función Signo

- Entradas:
  - Caracter: Variable tipo *char* de la que se desea saber si corresponde a un símbolo aritmético.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si el carácter introducido corresponde con un símbolo o no.

Funcionamiento

Sencillo funcionamiento que se basa en comparar el carácter introducido por valor con los símbolos aritméticos '+', '-', '*', '/', '%', '=', '<' y '>', para ver si se corresponde con alguno. En caso positivo devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función Numero

- Entradas:
  - Caracter: Variable tipo *char* de la que se desea saber si corresponde a un número.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si el carácter introducido corresponde con un número o no.

Funcionamiento

El funcionamiento es bastante simple, compara el carácter introducido por valor con los símbolos numéricos, para ver si se corresponde con alguno. En caso positivo devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función Letra

- Entradas:
  - Caracter: Variable tipo *char* de la que se desea saber si corresponde a una letra.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si el carácter introducido corresponde con una letra o no.

Funcionamiento

Se compara el carácter introducido por valor con los símbolos alfabéticos para ver si se corresponde con alguno. En caso positivo devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función LineaVacia

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si corresponde a una línea en blanco.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida esta en blanco o no.

Funcionamiento

Sencilla función que compara carácter a carácter hasta el final de línea para comprobar si en alguna de sus posiciones tiene introducido un símbolo diferente al espacio o al tabulador. En caso positivo devuelve el carácter 'N' indicando que no esta vacía, en caso contrario devuelve el carácter 'S' para indicar que realmente es una línea en blanco.

La función RestoLineaVacia

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si corresponde a una línea en blanco a partir de una determinada posición.
  - Pos: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual queremos saber si esta en blanco.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida esta en blanco o no a partir de la posición indicada.

Funcionamiento

Función muy similar a la anterior solo que esta vez se compara carácter a carácter hasta el final de línea comenzando desde una posición inicial para comprobar si en alguna de sus posiciones tiene introducido un símbolo diferente al espacio o al tabulador. En caso positivo devuelve el carácter 'N' indicando que no esta vacía, en caso contrario devuelve el carácter 'S' para indicar que a partir de esa posición es una línea en blanco.

La función ObtenCadenaRestante

- Entradas:
  - Linea: Variable tipo *char** de la que se desea obtener el tramo acotado entre una posición indicada y el final de la línea.
  - Pos: Variable tipo *int* que nos indicará la posición a partir de la cual comienza el tramo de línea el cual queremos obtener.
- Salidas:
  - Restante: Variable tipo *char** en la que se almacenará el tramo de línea que se nos va a devolver.

Funcionamiento

La función almacena es la variable 'Restante' la parte de la cadena 'Linea' que esta acotada entra la posición indicada en 'Pos y el final de la misma, independientemente de lo que haya almacenado en cada posición de esta y sin incluir la posición 'Pos'; es decir se nos devuelve lo que haya entre 'Pos+1' y el final de 'Linea'.

La función ThenAlcanzado

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "then".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "then" o no.

Funcionamiento

Esta función comprueba que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "then". Este tipo de funciones es útil en cuanto a búsqueda de diferentes estructuras en la lectura del código, para poder diferenciar sin problemas los comienzos o finales de los diferentes tipos de estructuras que podamos encontrar, ya sean *'if', 'while', 'for'*, etc... de este modo es como llevaré control de en que estructura me encuentro exactamente mientras leo el código. Se ha tenido en cuenta que el substring buscado ("then") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "then", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función ElseAlcanzado

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "else".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "else" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "else". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "else". Se ha tenido en cuenta que el substring buscado ("else") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "else", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función DoAlcanzado

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "do".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.

- Salidas:
    - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "do" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "do". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "do". Se ha tenido en cuenta que el substring buscado ("do") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "do", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función ToAlcanzado

- Entradas:
    - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "to".
    - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
    - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "to" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "to". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "to". Se ha tenido en cuenta que el

substring buscado ("to") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "to", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función DownToAlcanzado

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "downto".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "downto" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "downto". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "downto". Se ha tenido en cuenta que el substring buscado ("downto") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "downto", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función UntilAlcanzado

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "until".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "until" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "until". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "until". Se ha tenido en cuenta que el substring buscado ("until") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "until", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función OfAlcanzado

- Entradas:
  - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "of".
  - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.

- Salidas:
    - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "of" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "of". Comprobará que exactamente en la posición 'I' de la cadena de caracteres 'linea' comienza el string "of". Se ha tenido en cuenta que el substring buscado ("of") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "of", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función EndAlcanzado

- Entradas:
    - Linea: Variable tipo *char** de la que se desea saber si incluye en la posición indicada la palabra "end".
    - I: Variable tipo *int* que nos indicará la posición de la línea a partir de la cual debemos buscar.
- Salidas:
    - La función devuelve 'S' o 'N' dependiendo de si la línea introducida tiene en la posición indicada la palabra "end" o no.

Funcionamiento

Similar a la función anterior solo que esta vez la función buscará el substring "end". Comprobará que exactamente en la posición 'I' de la cadena

de caracteres 'linea' comienza el string "end". Se ha tenido en cuenta que el substring buscado ("end") pueda formar parte del nombre de una variable o del nombre de una función, por lo que se comprueba que se corresponda exactamente con la palabra reservada del compilador y no con otra cosa. Si en la posición 'I' de la cadena 'linea' comienza el substring "end", el cual se corresponde sin ninguna duda con esa palabra reservada del compilador la función devolverá el carácter 'S', en caso contrario devolverá el carácter 'N'.

La función EsCaracterValidoCondicion

- Entradas:
  - Caracter: Variable tipo *char* de la que se desea saber si puede formar parte una condición .
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si el carácter podría formar parte de una condición o no.

Funcionamiento

Función muy simple que evalúa si el carácter introducido puede formar parte de una condición del lenguaje Pascal o no. Se considera caracter válido a aquél que no es ni signo, ni espacio, ni parentesis; es decir, todos aquellos caracteres que puedan formar parte de una variable. En caso de que se considere que sí puede estar en una condición la función devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función EsVariableValida

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si se corresponde con un nombre de variable válida del lenguaje Pascal .

- Salidas:
    - La función devuelve 'S' o 'N' dependiendo de si toma la cadena introducida como el nombre de una variable válida o no.

Funcionamiento

Función que comprueba que la cadena introducida se corresponda con un posible nombre de variable válida en Pascal, es decir, que no contenga ninguna de los símbolos que según Pascal hace un nombre de variable erróneo; como podría ser que comience con un símbolo numérico o contenga un espacio en su interior. En el caso de que pudiera ser un nombre válido de variable devuelve el carácter 'S', en caso contrario devuelve el carácter 'N'.

La función Limp_Coment

- Entradas:
    - Línea: Variable tipo *char** de la que se eliminar los comentarios que pudiera tener.
    - C: Variable tipo *char** que indica a la función si ya se esta leyendo un comentario o no.
- Salidas:
    - La función devuelve mediante la variable 'C' si cuando se acabó la línea había un comentario abierto o no.

Funcionamiento

Función que elimina del código los comentarios que pudiera haber puesto el usuario, y que es utilizada cada vez que la aplicación lee una línea del código nueva. Se estudia desde el principio hasta el final toda la línea y en el caso de se abra un comentario se indica mediante una variable ('C'), que a su vez es devuelta para que en próximas llamadas a la función se pueda saber de antemano si se quedó un comentario abierto o no. La variable 'C' puede contener:

- No : indica que no existe comentario abierto.
- Sl : indica que se quedo abierto un comentario de llaves ({...}).
- Sp : indica que se quedó abierto un comentario de paréntesis ((*...*)).

De este modo controlamos en cualquier momento si se esta leyendo un comentario o líneas de código válidas. Cuando se cierra un comentario se elimina de la línea.

La función TomarNuevaLinea

- Entradas:
  - Esta función no tiene entradas.
- Salidas:
  - Línea: Variable tipo *char** que contiene una nueva línea de código fuente del fichero.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero del código fuente.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea y es necesario inicializar.

Funcionamiento

Esta función toma una nueva línea de código del fichero de entrada y la almacena en la variable 'linea'. Después se le eliminan los comentarios que pudiera tener y se pasan todos sus caracteres a minúsculas para que no haya posterior problemas de lectura. También inicializa la variable 'Pos' a 0, que es la variable mediante la cual se controla la posición de las variables a lo largo de una línea, y devuelve el número de línea, que es la posición que ocupa esa línea de código en el fichero de entrada y es útil almacenar para ubicar los errores mas tarde.

La función LimpiarBlancosIniciales

- Entradas:
  - Linea: Variable tipo *char** de la que se eliminar los espacios iniciales.
- Salidas:
  - Línea: La función devuelve la propia cadena de caracteres pero limpia de espacios iniciales.

Funcionamiento

Función muy sencilla que tomando una cadena de caracteres devuelve ese misma cadena pero después de haber eliminado todos los espacios en blanco que puedan existir desde el principio de la cadena hasta la primera posición con un símbolo distinto de el espacio o el tabulador.

La función TomarNuevaLinea

- Entradas:
  - Línea: Variable tipo *char** que contiene una nueva línea de código fuente del fichero.
- Salidas:
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero del código fuente.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea y es necesario inicializar.

Funcionamiento

Función similar a la anterior, solo que esta vez se comprueba que 'linea' sea una cadena de caracteres vacía; en caso positivo tomará una nueva

cadena del fichero de entrada modificando las variables pasadas por referencia necesarias (*NumLinea, Pos* , etc...).

La función EsPrincipioCodigo

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "begin".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "begin" o no.

Funcionamiento

Sencilla función que comprueba si al comienzo de la línea esta la palabra "begin" o no para saber si esa línea es el comienzo del código. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsFinCodigo

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "end".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "end" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "end" o no para saber si esa línea es el fin de un bloque de código. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsLlamUnidades

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "uses".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "uses" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "uses" o no para saber si esa línea es el comienzo de las llamadas a unidades de Pascal. Esto es para ignorar las siguientes líneas ya que en este proyecto no se tienen en cuenta las llamadas a otras unidades ni las funciones o procedimientos definidos en otra unidad. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsDefVariables

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "var".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "var" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "var" o no para saber si esa línea es el comienzo de una definición de variables. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraIf

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "if".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "if" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "if" o no para saber si en esa línea comienza una estructura if. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraElse

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "else".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "else" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "else" o no para saber si en esa línea comienza una estructura else. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraFor

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "for".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "for" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "for" o no para saber si en esa línea comienza una estructura for. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraRepeat

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "repeat".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "repeat" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "repeat" o no para saber si en esa línea comienza una estructura repeat. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraWhile

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "while".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "while" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea esta la palabra "while" o no para saber si en esa línea comienza una estructura while. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructuraCase

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "case".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "case" o no.

Funcionamiento

Función similar a la anterior que comprueba si al comienzo de la línea está la palabra "case" o no para saber si en esa línea comienza una estructura case. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsEstructura

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra reservada de alguna estructura,ya sea "if", "else", "for", etc....
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra reservada de alguna estructura o no.

Funcionamiento

Función que comprueba si al comienzo de la línea esta una palabra reservada de alguna estructura mediante las funciones anteriores para saber si en esa línea comienza o no alguna estructura. Se usarán las funciones EsEstructuraIf, EsEstructuraElse, etc... para realizar esta comprobación. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsDefTipos

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "type".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "type" o no.

Funcionamiento

Sencilla función que comprueba si al comienzo de la línea esta la palabra "type" o no para saber si esa línea es el comienzo de una definición de

tipos. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función EsDefProgram

- Entradas:
  - Línea: Variable tipo *char** de la que se desea saber si comienza con la palabra "program".
- Salidas:
  - La función devuelve 'S' o 'N' dependiendo de si encuentra al comienzo de la línea la palabra "program" o no.

Funcionamiento

Sencilla función que comprueba si al comienzo de la línea esta la palabra "programa" o no para saber si esa línea es el comienzo del programa y por lo tanto del código. En caso positivo la función devolverá el carácter ´S´, y en caso contrario devolverá el carácter 'N'.

La función LeerDefProgram

- Entradas:
  - Línea: Variable tipo *char** en la cual se indica el nombre del programa.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función trata con una cadena de caracteres ('linea') la cual se ha comprobado con anterioridad, mediante la función EsDefProgram, es la primera línea del código y en la que se define el nombre que tiene el

programa a probar. Esta función tan solo toma el nombre del programa y lo almacena en el objeto "Programa" de la clase Programa, siendo esta variable la que devuelve.

La función LeerDefVariables

- Entradas:
  - Línea: Variable tipo *char** que contiene la línea donde comienza la definición de las variables.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero de entrada donde esta el código.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función cuyas variables se están definiendo, ya sea el programa principal u otra función.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función identifica todas las variables que aparecen en una definición de variables leyendo línea a línea y sin tener en cuenta el tipo de la variable, tan solo tomando su nombre. Posteriormente almacena estas variables dentro del objeto CFuncion que corresponde, como se explicó anteriormente, usando para ello las funciones privadas de la clase CFuncion, a la cuales se accede gracias a que se pasa por referencia la variable "Programa" (objeto de la clase CPrograma).

La función LeerCondicion

- Entradas:
  - Línea: Variable tipo *char** que contiene la línea donde comienza la definición de las variables.
  - I: Variable tipo *int* que indica la posición de la línea a partir de la cual se debe empezar a leer la condición.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero de entrada donde esta el código.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la condición.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es la encargada de leer y analizar las condiciones que encontremos durante la lectura del código. La condición a tratar esta en la cadena de caracteres "linea" a partir de la posición "I". Se usa un algoritmo que identifica las variables así como también las posibles llamadas a funciones que podemos encontrar; en el caso de las variables se trata de una referencia por lo que se almacenará esa aparición ('R') en el objeto variable correspondiente de la clase CVariable usando una función privada de esta clase, mientras que en el caso de las llamadas a funciones se llamará a la función de análisis de estas, la cual se explicará mas adelante. La condición se comienza a leer como ya se ha dicho en la posición indicada mediante la variable 'I', pero se da por terminada cuando encuentra alguna de las palabras

reservadas, ya sea '*until', 'then'*, etc... es por eso que existen las funciones 'UntilAlcanzado', 'ThenAlcanzado', etc...

La función LeerCondicionRepeat

- Entradas:
  - Línea: Variable tipo *char** que contiene la línea donde comienza la definición de las variables.
  - I: Variable tipo *int* que indica la posición de la linea a partir de la cual se debe empezar a leer la condición.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero de entrada donde esta el código.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la condición.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
  - Condicion: Variable tipo *char** que almacena la condición leída.

Funcionamiento

Esta función es similar a la anterior solo que en este caso es usada específicamente para leer las condiciones de las estructuras 'repeat', ya que se deben tratar de forma diferente a las demás estructuras puesto que no comparten la misma terminación y los algoritmos de lectura y análisis que son aplicables a unas pueden no ser aplicables a otras. Se almacena en la variable

'Condicion' la condición que determina la iteración de la estructura 'repeat' y se devuelve para su posterior análisis mediante otra función.

La función LeerNombreFuncion

- Entradas:
  - Línea: Variable tipo *char** que contiene la cabecera de una función.
- Salidas:
  - Nombre: Variable tipo *char** que almacena el nombre de la función.

Funcionamiento

Esta función toma la cabecera de una función y devuelve tan solo el nombre ignorando el resto de los datos de la línea.

La función LeerCabeceraFuncion

- Entradas:
  - Línea: Variable tipo *char** que contiene la cabecera de la función.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero de entrada donde esta el código.
  - Nombre: Variable tipo *char** que contiene el nombre de la función.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función recibe la cabecera de la función y realiza diversas tareas:

- lee el nombre de la función y lo almacena en la variable 'Nombre', para posteriormente saber donde almacenar las variables que forman los parámetros de la cabecera.
- crea el objeto tipo CFuncion con el nombre anteriormente leído para posteriormente poder almacenar funciones.
- lee y analiza los parámetros para identificar las diferentes variables y poder almacenarlas junto con su tipo ('*valor*' o '*referencia*') en 'Programa', objeto de la clase CPrograma.

La función LeerSentencia

- Entradas:
  - Línea: Variable tipo *char** que contiene la sentencia a estudiar.
  - NumLinea: Variable tipo *int* que indica el número de línea que ocupa 'linea' en el fichero de entrada donde esta el código.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la sentencia.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es importante ya que es la encargada de leer las sentencias que podemos encontrar a lo largo del código del fichero de entrada. Primero identifica el tipo de sentencia, viendo si se trata de una estructura o no, en caso positivo se llamará a la función encargada de leer las

estructuras, en caso negativo se almacenará la sentencia tal cual en una variable para su posterior tratamiento mediante otra función que se explicará más adelante.

La función TratarSentencia

- Entradas:
  - Línea: Variable tipo *char** que contiene la sentencia a tratar.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la sentencia en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la sentencia.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es la encargada de tratar las sentencias leídas mediante la función anterior. Primero se encarga de estudiar que tipo de sentencia se trata, ya que puede ser una asignación o la llamada a alguna función; en el caso de que sea una asignación se mira cual es la variable a la cual hay que añadirle la aparición de una definición a su estructura 'variable' (objeto de la clase CVariable), y se estudia el resto de la asignación, por el contrario si se trata de la llamada a una función se leen y tratar los parámetros mediante la función "LeerParametros" que trataremos más adelante.

La función TratarCondicionFor

- Entradas:
  - Línea: Variable tipo *char** que contiene la condición a tratar.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la condición en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la condición.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
  - StrAux: Variable tipo *char** en la cual se almacenará el nombre de la variable que controla la condición de la estructura *for*.

Funcionamiento

Esta función lee y analiza las condiciones de las estructuras *for* que por sus características especiales se deben leer de modo diferente al resto de las demás estructuras, ya que la DrCadena que se saca de este tipo de condiciones tienen una forma muy especifica como se indicó en el apartado 2.3.1. Se compone de tres partes: primero se almacenará la variable de la condición en 'StrAux', después se leerá la parte de la asignación para por último añadir la definición a la variable almacenada en 'StrAux' mediante las funciones privadas de la clase CVariable.

La función LeerBloqueCodigo

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo del bloque de código.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta el comienzo del bloque de código en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la condición.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Primero debo decir que se entiende como bloque de código al conjunto de sentencias y estructuras que se encuentren entre un comienzo ("begin") y un fin ("end"). Esta función se encarga de leer sentencia a sentencia todas las líneas comprendidas entre esos dos puntos mediante la función específica para leer sentencias('LeerSentencia(...)') .

La función LeerEstructuraIf

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura "if".

- NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
- Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
- Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
- NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.

- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función puede ser considerada como de las más complejas e importantes de la aplicación, ya que lee totalmente de principio a fin una estructura "if", con todas las complicaciones y problemas que ello conlleva; pasaré a explicar más detenidamente paso a paso que hace esta función:

- primero se posiciona delante de la condición del "if" y mediante las funciones de lectura y análisis de condiciones vistas anteriormente se trata,
- después se busca la parte verdadera del "if" siendo esta una sentencia, otra estructura o un bloque de código, por lo que también se usan las funciones de antes ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura') para tratarlas, como se observa esta función puede ser recursiva ya que las funciones que tratan las sentencias pueden volver a llamar a 'LeerEstructuraIf'
- por último se comprueba si existe la parte falsa o parte del "else", en caso positivo se llamará a la función de 'LeerEstructuraElse' que más adelante se definirá.

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras, en las listas de apariciones de cada variable perteneciente a la función en cual está ubicado el "if".

La función LeerEstructuraElse

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura "else".
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.
  - EsDeCase: Variable tipo bool que indica si el "else" a tratar procede de un "if" o de un "case"
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función, como la anterior, es de las más importantes de la aplicación, ya que lee completamente una estructura "else", con todas las complicaciones y problemas que ello conlleva; esta función lee una sentencia, otra estructura o un bloque de código, y para analizarlo se usan las funciones de antes ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura'). Hay que decir que no es igual un "else" de una estructura "if" que el de una estructura "case", por ello la variable que indica de donde proviene.

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras que pueda haber, en las listas de apariciones de cada variable perteneciente a la función en la cual está ubicado el "else".

La función LeerEstructuraFor

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura "for".
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la funcion en la cual se encuentra la estructura.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función, como la anterior, es muy importante, ya que lee completamente una estructura "for"; realiza paso a paso las siguientes tareas:

- lee la condición del "for", y ya que es un caso especial necesita una función específicamente para ello tal y como se explicó antes, se almacena totalmente la condición y se llama a 'TratarCondicionFor' para analizarla,

- después se comprueba si la iteración de la estructura se compone solo de una sentencia, de otra estructura o de un bloque de código y se llama a las funciones pertinentes para tratarlo ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura'), como se observa esta función puede ser recursiva ya que las funciones que tratan las sentencias pueden volver a llamar a 'LeerEstructuraFor'.

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras que pueda haber, en las listas de apariciones de cada variable perteneciente a la función en la cual está ubicado el "for".

La función LeerEstructuraWhile

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura "while".
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función también es de las más importantes ya que lee y analiza de principio a fin una estructura "while", realiza paso a paso las siguientes tareas:

- busca y lee la condicion del "while", almacena la condición en una variable tipo *char** y se llama a la función encargada de tratarla, que en este caso es "LeerCondicion"

- después se posiciona en el comienzo del código iterativo de la estrcutura, y dependiendo de si es una sentencia, otra estructura o un bloque de código llama a una función o a otra ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura'), como se observa esta función puede ser recursiva ya que las funciones que tratan las sentencias pueden volver a llamar a 'LeerEstructuraWhile'.

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras que pueda haber, en las listas de apariciones de cada variable perteneciente a la función en la cual está ubicado el "while".

La función LeerEstructuraRepeat

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura "repeat".
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.

- Salidas:
    - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es similar a la anterior y también importante ya que lee y analiza las estructuras "repeat". Estos son los que sigue:

- se posiciona en el comienzo del código iterativo de la estructura, y dependiendo de si es una sentencia, otra estructura o un bloque de código llama a una función o a otra ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura'),esto se repite hasta que encuentra el 'until'. Como se observa esta función puede ser recursiva ya que las funciones que tratan las sentencias pueden volver a llamar a 'LeerEstructuraRepeat',

- después se posiciona en la condición de la estructura iterativa y como anteriormente se trata mediante la función 'LeerCondicion'

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras que aparezcan, en las listas de apariciones de cada variable perteneciente a la función en la cual está ubicado el "repeat".

La función LeerEstructuraCase

- Entradas:
    - Línea: Variable tipo *char** que contiene el comienzo de la estructura "case".
    - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.

- Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
- Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
- NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.

- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es similar a 'LeerEstrcuturaIf' ya que en cierto modo la estructura 'if' y la estructura 'case' son similares. Estos son los que sigue:

- se posiciona en la condición y la lee mediante la función 'LeerCondición',

- después se posiciona en la que podríamos llamar al primer camino del 'case' y dependiendo de si hay una sentencia, otra estructura o un bloque de código llama a una función o a otra ('LeerSentencia', 'LeerBloqueCodigo' o 'LeerEstructura'),esto se repite sucesivas veces examinando los diversos caminos del 'case' hasta que encuentra el 'else' o el fin de la estructura. Como se observa esta función puede ser recursiva ya que las funciones que tratan las sentencias pueden volver a llamar a 'LeerEstructuraCase',

- después dependiendo de si tiene 'else' o no, y para finalizar, llamará a 'LeerEstructuraElse'.

Es necesario decir que mientras se hacen todas estas tareas se indican los comienzos y finales, de las estructuras que aparezcan durante el 'case', en

las listas de apariciones de cada variable perteneciente a la función en la cual está ubicado este "case".

La función LeerEstructura

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la estructura.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la estructura en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - NombreFuncion: Variable tipo *char** que contiene el nombre de la función en la cual se encuentra la estructura.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es bastante más simple que las anteriores ya que lo único que hace son dos pasos:

- comprueba que tipo de estructura se trata usando las funciones explicadas anteriormente ('EsEstructuraIf', 'EsEstructuraWhile', etc...),

- llama a la función indicada para tratar esa estructura, estas funciones también se han visto antes ('LeerEstructuraIf', 'LeerEstructuraCase', etc ...).

Como se observa esta función puede ser recursiva ya puede alguna de las funciones que leen específicamente un tipo de estructura ('LeerEstructuraIf', 'LeerEstructuraCase', etc...) puede volver a llamar a 'LeerEstructura'.

La función LeerFuncion

- Entradas:
  - Línea: Variable tipo *char** que contiene el comienzo de la función, es decir, la cabecera.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la función en el fichero de entrada.
  - Comentario: Variable tipo *char** necesaria para la función de limpieza de comentarios.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es, sin duda, una de las más importantes y extensas de la aplicación ya que se encarga de leer y estudiar una función desde el principio (su cabecera) hasta el final (el final del bloque de código). Pasaré a resumir que hace paso a paso:

lo primero que hace es leer la cabecera de la función, para saber el nombre de esta y almacenar los parámetros con su tipo en la lista de funciones formada por los objetos de la clase CFuncion,

después pasa a comprobar si tiene definición de variables buscando 'var' (variables locales), y en caso positivo lee y almacena estas variables ('LeerDerVariables'), como anteriormente se explicó, en la lista de funciones formada por los objetos de la clase CFuncion,

inmediatamente después comprueba si existe funciones definidas dentro de esta, en ese caso se llamaría recursivamente para obtener todos los datos,

para finalizar con las variables añade a esta función todas las variables locales de su función superior para que sean tratadas como globales, es decir, si la función que estamos leyendo es definida desde el programa principal las variables del programa principal se le meterían a esta función como globales, esto se explica en la siguiente función,

por último se lee el código de la función propiamente dicho, desde el 'begin' hasta el 'end' se leen todas sus sentencias y estructuras mediante la función explicada antes 'LeerBloqueCódigo'.

Cuando se finaliza la lectura del código se comprueban todas las variables globales que le hemos metido y aquellas que no hayan sufrido ninguna modificación durante la lectura de la función, entendiendo modificación como una referencia o una definición, son borradas de la lista de variables de esa función.

La función AnadirVariablesGlobales

- Entradas:
  - Función: Variable tipo *char** que contiene el nombre del objeto de la clase CFuncion al cual hay que añadirle las variables.
- Salidas:

- Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta sencilla función es la encargada de añadir a una función cualquiera las variables que puede usar como globales, es decir, las variables locales de la función que la engloba. Será mas ilustrativo mostrarlo con un ejemplo: si la función 'Prueba' esta definida en el programa principal a esta función se le meterán como globales todas las locales del programa principal, pero si dentro de 'Prueba' esta definida la función 'Prueba2' a esta última función se le deberán añadir como globales todas las variables de 'Prueba'.

Esto se hace porque a priori no podemos saber que variables se van a usar en una función como globales, por eso se añaden todas, para después eliminar aquellas que no han sido usadas.

Estas variables se añaden normalmente como cualquier otra, a la lista de variables que tenga la estructura de la función que estamos tratando (objeto de la clase CFuncion), pero con la diferencia de que su tipo es 'global'.

La función BorrarGlobales_sin_usar

- Entradas:
  - Función: Variable tipo *char** que contiene el nombre del objeto de la clase CFuncion al cual hay que comprobarle las variables globales.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es la encargada de realizar la tarea, indicada en la definición de función anterior, de eliminar de la lista de variables de un objeto de la clase CFuncion la variables con tipo 'global' que no han sido usadas, ni definidas ni referenciadas. Tan solo comprueba una a una todas las variables globales de esa función para eliminar aquellas sin usar.

La función LeerParametros

- Entradas:
  - Línea: Variable tipo *char** que contiene la parte de los parámetros de la cabecera de una definición de función.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la cabecera de la función en el fichero de entrada.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - FuncionLlamada: Variable tipo *char** que contiene el nombre de la función llamada, cuyos parámetros vamos a estudiar.
  - FuncionActual: Variable tipo *char** que contiene el nombre de la función desde la cual se llama a la otra función, es decir, la función en cuyo código se encuentra la llamada a la función 'FuncionLlamada'.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función es la más importante en lo que se refiere a la lectura de las llamadas a las funciones, ya que no solo lee y trata los parámetros, sino que es donde se genera la estructura necesaria para almacenar todos los datos de una llamada de este tipo, o lo que es lo mismo, genera el objeto

'Cabecera' de la clase CCabecera desde el cual más tarde partirán las listas de parámetros y posiciones (de cabecera parte una lista de parámetros, de cada parámetro parte una lista de posiciones y de cada posición parte una lista de apariciones). Explicaré mas detenidamente que hace esta función:

- primero lee cada parámetro uno a uno de 'Linea' y los almacena en una lista de parámetros que tiene el objeto 'Cabecera' (objeto de la clase CCabecera),

- a cada parámetro se le asigna una o varias posiciones, esto es, si cada variable solo aparece una vez como parámetro (Prueba(x,y)) cada parámetro tendrá en su lista de posiciones solo una posición ('x' tendrá la posición 1 , 'y' tendrá la posición 2), sin embargo si alguna variable aparece varias veces en la cabecera (Prueba(x,x)) el parámetro tendrá varias nodos en su lista de posiciones ('x' tendrá las posiciones 1 y 2),

- después se estudia cada parámetro por separado, pudiendo darse dos casos:
  - si el parámetro era pasado por valor se le asigna una 'R' a su lista de apariciones que tiene esa variable en esa función en el objeto 'Cabecera',
  - si el parámetro era pasado por referencia se le asigna la DrCadena correspondiente a la variable de la función llamada que en la definición estaba ubicada en esa posición de la cabecera.

- por último se examinan las globales para comprobar si alguno de los parámetros además esta como global para unir ambas DrCadenas.

Una vez que se han tratado todos los parámetros y se han añadido las cadenas necesarias a la variables de la función que aparecen en la llamada de la función que estudiamos, se elimina la estructura que almacenaba toda esa información (el objeto de la clase CCabecera).

La función CompletarListaParametros

- Entradas:
  - FuncionActual: Variable tipo *char** que contiene el nombre de la función la cual engloba a la función llamada.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la cabecera de la función en el fichero de entrada.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
- Salidas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.

Funcionamiento

Esta función es la que se encarga de realizar la tarea de asignar a cada posición de cada parámetro la lista de apariciones que corresponda a las variables ubicadas en esas mismas posiciones en la cabecera de la definición de la función. Se ilustra mediante un ejemplo:

Tenemos la definición de la función 'Prueba':

```
Procedure Prueba(var x:integer;y:char);
```

Y la llamada a la función:

```
Prueba(a,b);
```

Por lo tanto al haberse realizado la llamada a la función, tendríamos que asignar a la variable 'a' la lista de apariciones de la variable 'x', ya que esta pasada por referencia.

La función SacaDRCadenas_Parametros

- Entradas:
  - FuncionActual: Variable tipo *char** que contiene el nombre de la función la cual engloba a la función llamada.
  - NumLinea: Variable tipo *int* que indica el número de línea en la que esta la cabecera de la función en el fichero de entrada.
  - Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
- Salidas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.

Funcionamiento

Esta función es la que se encarga de sacar las DrCadenas de los parámetros concatenando los nodos de la lista de apariciones (lista objetos 'LineaVar') de cada uno. Más tarde añadirá esa cadena (la DrCadena concatenada) como una aparición más de la variable que fue pasada como parámetro.

La función TratarGlobales

- Entradas:
  - FuncionActual: Variable tipo *char** que contiene el nombre de la función la cual engloba a la función llamada.
  - FuncionLlamada: Variable tipo *char** que contiene el nombre de la función llamada.

- NumLinea: Variable tipo *int* que indica el número de línea en la que esta la cabecera de la función en el fichero de entrada.
- Pos: Variable tipo *int* que controla las posiciones de las variables en la línea.
- Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

- Salidas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.

Funcionamiento

Esta función es la que se encarga de buscar las variables globales de 'FuncionLlamada' que no han sido eliminadas, y se tratan para después poder unir sus cadenas con las variables que corresponda de las que han sido pasadas como parámetros.

La función Posiciones_Vacias

- Entradas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.
- Salidas:
  - Esta función devuelve 'S' o 'N' dependiendo de si su lista de posiciones es vacía o no.

Funcionamiento

Esta función simplemente comprueba si la lista de posiciones de un parámetro esta vacía o no. En caso positivo devolverá el carácter 'S', y en caso contrario el carácter 'N'.

La función Elimina_todas_PrimeraLineaVar

- Entradas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.
- Salidas:
  - Cabecera: Objeto de la clase CCabecera en la cual se almacenan todos los datos concernientes a los parámetros en la llamada a una función.

Funcionamiento

Función auxiliar muy sencilla que tan solo elimina el primer elemento de la lista de apariciones (objetos de la clase CLineaVar), de cada posición de cada parámetro.

La función LeerPrograma

- Entradas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
  - NombreFichero: Variable tipo *char** que contiene el nombre del fichero de entrada de la aplicación.
- Salidas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.

Funcionamiento

Esta función puede ser considerada como la más importante en lo que se refiere a la lectura del código. Realiza diversas tareas importantes que paso a enumerar:

primero se abre el fichero de entrada para lectura, comprobándose que se abre correctamente,

después se lee y almacena el nombre del programa mediante la función explicada anteriormente "LeerDefProgram",

más tarde busca y trata todas las variables del programa, es decir el programa principal, para almacenarlas y posteriormente añadir su lista de apariciones (objetos de la clase CLineaVar), mediante la función 'LeerDefVariables'

después busca, lee y analiza todas las funciones que puedan estar definidas antes del código del programa principal mediante la función 'LeerFunción',

por último se lee el bloque de código que compone el programa principal sentencia a sentencia y estructura a estructura ('LeerBloqueCodigo') hasta el final del código, y por lo tanto del programa.

5.1.3.2 Funciones de análisis

Estas son las funciones de la aplicación que se encargan de tareas tales como obtener las DRCadenas de la estructura de datos, tratar las DrCadenas, obtener los datos necesarios de las mismas para poder analizarlas, analizar los datos, mostrar los errores, y otras tareas menores.

En la figura 5.7. se muestra las llamadas entre las funciones más importantes englobadas dentro del Bloque2 mediante las cuales analizan y muestra los errores.

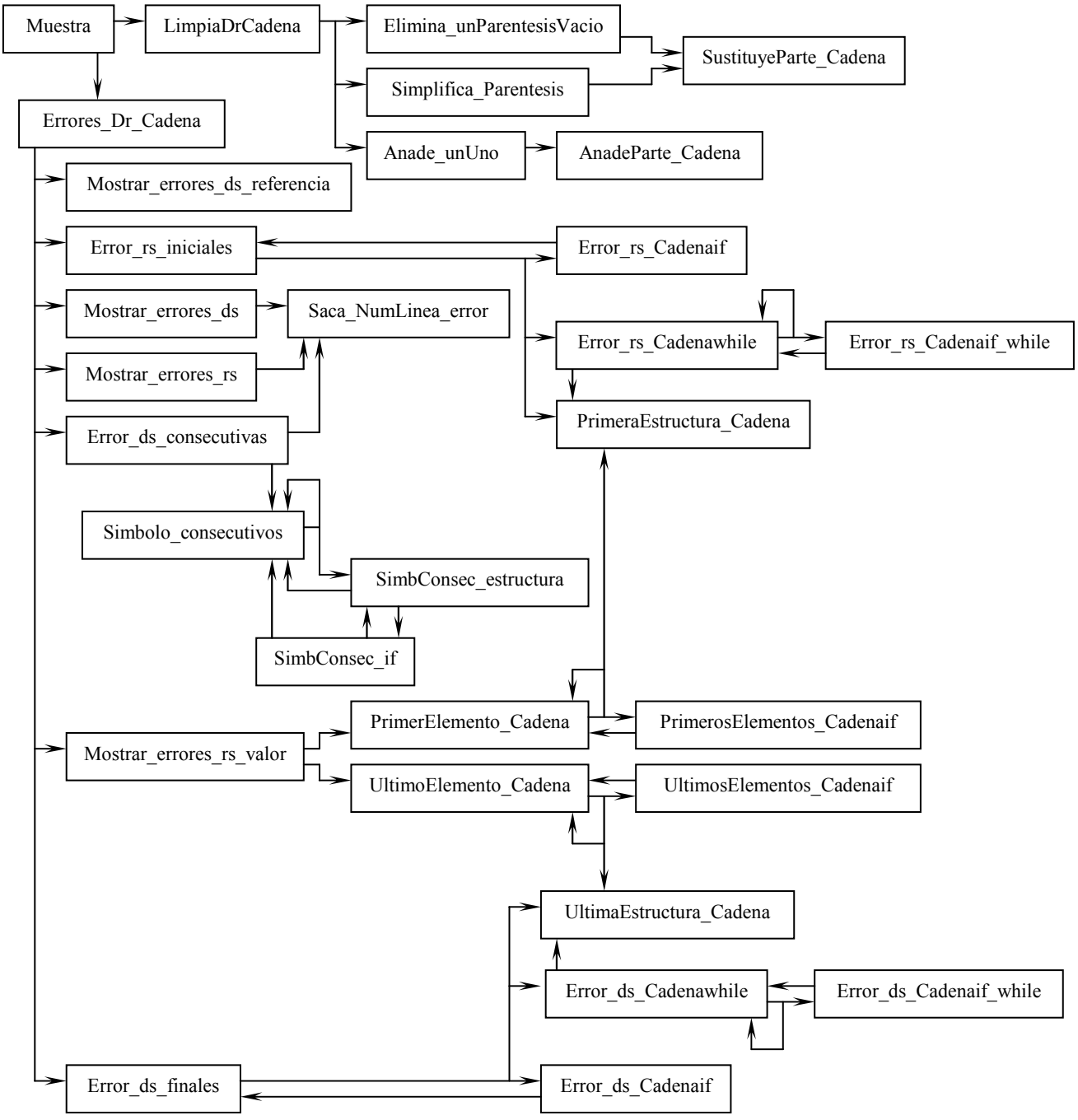


Figura 5.7: Esquema de llamadas de funciones de análisis de las DrCadenas.

A continuación se explicarán una a una y más detenidamente las funciones que englobé en el segundo bloque o también llamadas de tratamiento y análisis de las DrCadena, y salida de los errores.

La función SustituyeParte_Cadena

- Entradas:
  - Cadena: Variable tipo *char** de la que se desea cambiar un trozo de la misma por otra cadena de caracteres.
  - CaracIni: Variable tipo *int* que nos indica la posición de inicio del intervalo que debemos sustituir en la cadena 'Cadena'.
  - CaracFin: Variable tipo *int* que nos indica la posición de fin del intervalo que debemos sustituir en la cadena 'Cadena'.
  - Parte: Variable tipo *char** que contiene la cadena de caracteres que vamos a introducir en 'Cadena' entre las dos posiciones indicadas.
- Salidas:
  - Cadena: Variable tipo *char** modificada.

Funcionamiento

Función auxiliar para el tratamiento de las cadenas de caracteres. Sustituye la cadena introducida 'Parte' por el intervalo de 'Cadena' que esta comprendido entre las dos posiciones indicadas ('CaracIni', 'CaracFin').

La función AnadeParte_Cadena

- Entradas:
  - Cadena: Variable tipo *char** de la que se desea cambiar un trozo de la misma por otra cadena de caracteres.
  - Parte: Variable tipo *char** que contiene la cadena de caracteres que vamos a añadir a la variable 'Cadena'.

- Pos: Variable tipo *int* que nos indica la posición de 'Cadena' a partir de la cual se añade la cadena de caracteres 'Parte'.

- Salidas:
  - Cadena: Variable tipo *char** modificada.

Funcionamiento

Función auxiliar para el tratamiento de las cadenas de caracteres. Añade la cadena introducida 'Parte' a partir de la posición 'Pos' en la cadena de caracteres 'Cadena'; el resto de la cadena original que quedaba comprendido entre la posición indicada y el final de la misma se concatena al final de la nueva cadena.

La función TipoParentesis

- Entradas:
  - Cadena: Variable tipo *char** a la cual se desea examinar.
  - Posicion: Variable tipo *int* que señala la posición de 'Cadena' en la cual esta el paréntesis abierto que indica el comienzo de la estructura.
- Salidas:
  - PosParenFin: Variable tipo *int* que señala la posición de 'Cadena' en la cual termina la estructura.
  - La función devuelve mediante una cadena de caracteres el tipo de la estructura, si era *'while', 'for', 'if'*, etc.

Funcionamiento

Sencilla función que recibe una cadena de caracteres (una DrCadena) y la posición en la cual comienza una estructura en la misma, busca el final de la estructura, es decir, el paréntesis que la cierra, devuelve la posición en la cual esta ubicado e indica el tipo de estructura que era comprobando su terminación.

La función Elimina_unParentesisVacio

- Entradas:
  - Cadena: Variable tipo *char** a la cual se desea tratar.
- Salidas:
  - Modificada: Variable tipo *bool* que indica si la variable 'Cadena' fue modificada o no.

Funcionamiento

Función que trata una cadena de caracteres, que se corresponde con una DrCadena, para buscar símbolos innecesarios; en este caso se buscan paréntesis vacíos que son inútiles, como por ejemplo en la cadena 'DR(**()**o(D))', en el caso de que encuentre alguno se modifica la cadena mediante la rutina 'SustituyeParte_Cadena' y se indica que la cadena ha sido modificada al devolver la variable 'Modificada'.

La función Anade_unUno

- Entradas:
  - Cadena: Variable tipo *char** a la cual se desea tratar.
- Salidas:
  - Modificada: Variable tipo *bool* que indica si la variable 'Cadena' fue modificada o no.

Funcionamiento

Función que trata una cadena de caracteres, que se corresponde con una DrCadena, para añadir los símbolos '1' necesarios para indicar que en una parte, ya sea la verdadera o la falsa, de un 'if' o un 'case' no aparece la variable. Busca paréntesis vacíos dentro de estas estructuras y en el caso de que los halle añade el símbolo y se indica que la cadena ha sido modificada al devolver la variable 'Modificada'.

La función Simplifica_Parentesis

- Entradas:
  - Cadena: Variable tipo *char** a la cual se desea tratar.
- Salidas:
  - Simpli: Variable tipo *bool* que indica si la variable 'Cadena' fue modificada o no.

Funcionamiento

Función que trata una cadena de caracteres, que se corresponde con una DrCadena, para arreglar las posibles estructuras que solo tengan un símbolo, es decir, si la cadena tiene 'D(R)*' esta función la deja como 'DR*'. En el caso de que sea modificada la cadena se indica mediante la variable 'Simpli'.

La función LimpiaDrCadena

- Entradas:
  - Cadena: Variable tipo *char** a la cual se desea limpiar.
- Salidas:
  - La función devuelve la variable 'Cadena' una vez se ha tratado.

Funcionamiento

Esta función es la que se encarga de la limpieza de las DrCadenas para su posterior análisis, es decir, se encarga de eliminar símbolos sobrantes, estructuras vacías, etc..

Para ello utiliza las funciones descritas anteriormente: 'Elimina_unParentesisVacio', 'Anade_unUno' y 'SimplificaParentesis'. Se le aplica a la cadena estas funciones tantas veces como haga falta y se devuelve una vez tratada.

La función CopiaFinParen

- Entradas:
  - Cadena: Variable tipo *char** de la cual se desea obtener un fragmento.
  - Pos: Variable tipo *int* que indica la posición de la cadena en la cual comienza la estructura a devolver
- Salidas:
  - StrCp: Variable tipo *char** que almacena el fragmento de la cadena de caracteres 'Cadena' que se desea obtener.

Funcionamiento

Esta función auxiliar recibe una cadena de caracteres, que se corresponde con una DrCadena, y una posición de esta en la cual está un paréntesis abierto indicando el comienzo de una estructura. La función copia en la variable 'StrCp' la subcadena englobada desde la posición indicada hasta el final de la estructura, el cierre del paréntesis, y la devuelve.

La función QuitaParen_Estruc

- Entradas:
  - Cadena: Variable tipo *char** que contiene una estructura.
- Salidas:
  - La función devuelve la variable 'Cadena' una vez ha sido modificada.

Funcionamiento

Función muy sencilla que tan solo elimina los paréntesis inicial y final de la estructura contenida en la variable 'Cadena'.

La función PrimeraEstructura_Cadena

- Entradas:
  - Cadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - La función devuelve una cadena de caracteres, la cual contiene la primera estructura de la DrCadena de 'Cadena'.

Funcionamiento

Esta función es importante ya que analiza un DrCadena, contenida en la variable 'Cadena', y obtiene el primer símbolo de esta; en el caso de que en vez de un símbolo al inicio tenga una estructura la devolverá como si de un símbolo se tratase desde el paréntesis inicial hasta el paréntesis final. La función comprueba el primer elemento de la cadena y en el caso de que sea '(' busca el paréntesis de cierre que lo equilibre y devuelve toda la estructura, en caso contrario tan solo devuelve ese elemento.

La función UltimaEstructura_Cadena

- Entradas:
  - Cadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - La función devuelve una cadena de caracteres, la cual contiene la última estructura de la DrCadena de 'Cadena'.

Funcionamiento

Esta función es importante ya que analiza un DrCadena, contenida en la variable 'Cadena', y obtiene el última símbolo de esta; en el caso de que en vez de un símbolo al final tenga una estructura la devolverá como si de un símbolo se tratase desde el paréntesis inicial hasta el paréntesis final. La

función comprueba el último elemento de la cadena y en el caso de que sea ')' busca el paréntesis de inicio que lo equilibre y devuelve toda la estructura, en caso contrario tan solo devuelve ese elemento.

La función PrimerElemento_Cadena

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - StrElem: Variable tipo *char** que contiene todos los símbolos que son posibles símbolos iniciales de la DrCadena.

Funcionamiento

Esta función devuelve en una cadena de caracteres y separados por espacios todos los símbolos que puedan ser iniciales en la cadena, ya que es viable que en una DrCadena existan varios símbolos iniciales, por ejemplo, en el caso de que esta comience por una estructura *'if'*. En los casos de que al inicio de la cadena haya una estructura en vez de un símbolo se tomaría esta estructura y se la trataría como una nueva DrCadena, y así sucesivamente.

La función UltimoElemento_Cadena

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - StrElem: Variable tipo *char** que contiene todos los símbolos que son posibles símbolos finales de la DrCadena.

Funcionamiento

Esta función devuelve en una cadena de caracteres y separados por espacios todos los símbolos que puedan ser finales en la cadena, ya que es

viable que en una DrCadena existan varios símbolos finales, por ejemplo, en el caso de que esta termine con una estructura *'if'*. En los casos de que al fin de la cadena haya una estructura en vez de un símbolo se tomaría esta estructura y se la trataría como una nueva DrCadena, y así sucesivamente.

La función PrimerosElementos_Cadenaif

- Entradas:
  - DrCadena: Variable tipo *char*,* esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' en la cadena de caracteres 'DrCadena'.
- Salidas:
  - StrElem: Variable tipo char* en la que se almacenan todos los posibles símbolos iniciales de la DrCadena.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos iniciales de una Drcadena. Esta función se encarga de separar los símbolos iniciales de cada una de las partes de una estructura 'if' o 'case' para devolverlos mediante la variable 'StrElem'.

La función UltimosElementos_Cadenaif

- Entradas:
  - DrCadena: Variable tipo *char*,* esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' en la cadena de caracteres 'DrCadena'.
- Salidas:
  - StrElem: Variable tipo *char** en la que se almacenan todos los posibles símbolos finales de la DrCadena.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos finales de una DrCadena. Esta función se encarga de separar los símbolos finales de cada una de las partes de una estructura 'if' o 'case' para devolverlos mediante la variable 'StrElem'.

La función Error_rs_Cadenaif

- Entradas:
  - DrCadena: Variable tipo *char*,* esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' o 'case' en la cadena de caracteres 'DrCadena'.
  - NumR: Variable tipo *int* usada de contador, contabiliza los símbolos 'R' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrRI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'if' y 'case' de la DrCadena que pueden ser iniciales.
  - StrRW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que pueden ser iniciales.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos 'R' iniciales de una DrCadena. Esta función se encarga de separar las 'R' iniciales de cada una de las partes de una estructura 'if' o 'case' para devolver su posición mediante la variable 'StrRI'. Cada vez que se encuentra una 'R' se incrementa el contador 'NumR' para controlar en todo momento la posición de cada símbolo.

La función Error_rs_Cadenawhile

- Entradas:
  - DrCadena: Variable tipo *char*,* esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'while', 'for' o 'repeat' en la cadena de caracteres 'DrCadena'.
  - NumR: Variable tipo *int* usada de contador, contabiliza los símbolos 'R' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrRI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'if' y 'case' de la DrCadena que pueden ser iniciales.
  - StrRW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que pueden ser iniciales.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos 'R' iniciales de una DrCadena. Esta función se encarga de separar las 'R' iniciales de las estructuras 'while', 'for' o 'repeat' para devolver su posición mediante la variable 'StrRW'. Cada vez que se encuentra una 'R' se incrementa el contador 'NumR' para controlar en todo momento la posición de cada símbolo.

La función Error_rs_Cadenaif_while

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' o 'case' en la cadena de caracteres 'DrCadena'.

- NumR: Variable tipo *int* usada de contador, contabiliza los símbolos 'R' que van apareciendo a lo largo de la DrCadena.

- Salidas:
  - StrRI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'if' y 'case' de la DrCadena que pueden ser iniciales.
  - StrRW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que pueden ser iniciales.

Funcionamiento

Esta función es similar a rutina anterior 'Error_rs_Cadenaif' solo que en esta ocasión se trata el caso específico de que nos encontremos una estructura tipo 'if' o 'case' dentro de otra estructura tipo 'repeat', 'while' o 'for'. Esto es porque si nos encontramos algo de este tipo la estructura tipo 'if' no puede ser leída como antes, sino que debe ser considerada iterativa debido a que esta englobada dentro de otra que sí que es iterativa. Este caso no se controla con la función 'Error_rs_Cadenaif', pero por lo demás es igual, devuelve las 'R' mediante la variable 'StrRI' y cada vez que se encuentra una 'R' se incrementa el contador 'NumR' para controlar en todo momento la posición de cada símbolo.

La función Error_rs_iniciales

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - NumR: Variable tipo *int* usada de contador, contabiliza los símbolos 'R' que van apareciendo a lo largo de la DrCadena.
- Salidas:

- StrRI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'if' y 'case' de la DrCadena que pueden ser iniciales.
- StrRW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que pueden ser iniciales.

Funcionamiento

Esta función recibe una DrCadena y se encarga de identificar todas las 'R' de la cadena que pueden ser símbolo inicial de esta. En el caso de que al inicio en vez de un símbolo simple haya una estructura la función se vale de las rutinas 'Error_rs_Cadenaif' y 'Error_rs_Cadenawhile' para identificar a su vez los símbolos iniciales de esa estructura. Las 'R' que sean iniciales de la cadena se devuelven mediante las variables 'StrRI' y 'StrRW' en la cuales se indica la posición que ocupan. En el caso de que no existiese ningún error de este tipo en la cadena 'StrRI' y 'StrRW' se devolverían como cadenas vacías.

La función Error_ds_Cadenaif

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' o 'case' en la cadena de caracteres 'DrCadena'.
  - NumD: Variable tipo *int* usada de contador, contabiliza los símbolos 'D' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrDI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'if' y 'case' de la DrCadena que puedan ser finales.

- StrDW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que puedan ser finales.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos 'D' finales de una DrCadena. Esta función se encarga de separar las 'D' finales de cada una de las partes de una estructura 'if' o 'case' para devolver su posición mediante la variable 'StrDI'. Cada vez que se encuentra una 'D' se incrementa el contador 'NumD' para controlar en todo momento la posición de cada símbolo.

La función Error_ds_Cadenawhile

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'while', 'for' o 'repeat' en la cadena de caracteres 'DrCadena'.
  - NumD: Variable tipo *int* usada de contador, contabiliza los símbolos 'D' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrDI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'if' y 'case' de la DrCadena que puedan ser finales.
  - StrDW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que puedan ser finales.

Funcionamiento

Función auxiliar en la tarea de obtener los símbolos 'D' finales de una DrCadena. Esta función se encarga de separar las 'D' finales de las estructuras

'while', 'for' o 'repeat' para devolver su posición mediante la variable 'StrDW'. Cada vez que se encuentra una 'D' se incrementa el contador 'NumD' para controlar en todo momento la posición de cada símbolo.

La función Error_ds_Cadenaif_while

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - PosAux: Variable tipo *int* que indica el comienzo de una estructura 'if' o 'case' en la cadena de caracteres 'DrCadena'.
  - NumD: Variable tipo *int* usada de contador, contabiliza los símbolos 'D' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrDI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'if' y 'case' de la DrCadena que puedan ser finales.
  - StrDW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que puedan ser finales.

Funcionamiento

Esta función es similar a rutina anterior 'Error_ds_Cadenaif' solo que en esta ocasión se trata el caso específico de que nos encontremos una estructura tipo 'if' o 'case' dentro de otra estructura tipo 'repeat', 'while' o 'for'. Esto es porque si nos encontramos algo de este tipo la estructura tipo 'if' no puede ser leída como antes, sino que debe ser considerada iterativa debido a que esta englobada dentro de otra que sí que es iterativa. Este caso no se controla con la función 'Error_ds_Cadenaif', pero por lo demás es igual, devuelve las 'D' mediante la variable 'StrDI' y cada vez que se encuentra una 'D' se incrementa el contador 'NumD' para controlar en todo momento la posición de cada símbolo.

La función Error_ds_finales

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - NumR: Variable tipo *int* usada de contador, contabiliza los símbolos 'D' que van apareciendo a lo largo de la DrCadena.
- Salidas:
  - StrDI: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'if' y 'case' de la DrCadena que puedan ser finales.
  - StrDW: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que puedan ser finales.

Funcionamiento

Esta función recibe una DrCadena y se encarga de identificar todas las 'D' de la cadena que puedan ser símbolo final de esta. En el caso de que al final en vez de un símbolo simple haya una estructura la función se vale de las rutinas 'Error_ds_Cadenaif' y 'Error_ds_Cadenawhile' para identificar a su vez los símbolos finales de esa estructura. Las 'D' que sean finales de la cadena se devuelven mediante las variables 'StrDI' y 'StrDW' en la cuales se indica la posición que ocupan. En el caso de que no existiese ningún error de este tipo en la cadena 'StrDI' y 'StrDW' se devolverían como cadenas vacías.

La función UneCadenas_errores

- Entradas:
  - ErroresIf: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' y las 'R' de las estructuras 'if' y 'case' de la DrCadena que sean erróneas.

- ErroresWhile: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' y las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que sean erróneas.

- Salidas:
  - Str: Variable tipo *char** que contiene las cadenas 'ErroresIf' y 'ErroresWhile' concatenadas de un modo correcto.

Funcionamiento

Esta sencilla función se encarga de unir la cadena de errores de las estructuras 'while' con la cadena de errores de las estructuras 'if' en una tercera cadena llamada 'Str'. Comprueba que si existe un error que este en ambas cadenas, solo se meta una vez en 'Str'.

La función Mostrar_errores_rs

- Entradas:
  - ErroresIf: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'if' y 'case' de la DrCadena que sean erróneas.
  - ErroresWhile: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'R' de las estructuras 'while', 'for' y 'repeat' de la DrCadena que sean erróneas.
  - LineaVar: Variable tipo CLineaVar* usada para obtener la línea del código en la que esté una 'R' errónea.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función se encarga de mostrar por pantalla los errores tipo a) o de referencia inicial que pueda tener una determinada cadena, la cual ha sido examinada previamente por la función 'Error_rs_iniciales'. Para ello comprueba las cadenas 'ErroresIf' y 'ErroresWhile' que contienen las posiciones de las 'R'

que producen ese error, y por cada posición que encuentre mostrará por pantalla ese número y buscará en que línea de código se encuentra esa 'R' para también mostrar el número de línea.

La función Mostrar_errores_ds

- Entradas:
  - Errores: Variable tipo *char** que almacena, separadas por comas, las posiciones de las 'D' de la DrCadena que sean erróneas.
  - LineaVar: Variable tipo CLineaVar* usada para obtener la línea del código en la que esté una 'D' errónea.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función se encarga de mostrar por pantalla los errores tipo c) o de definición final que pueda tener una determinada cadena, la cual ha sido examinada previamente por la función 'Error_ds_finales'. Para ello comprueba la cadena 'Errores' que contiene las posiciones de las 'D' que producen ese error, y por cada posición que encuentre mostrará por pantalla ese número y buscará en que línea de código se encuentra esa 'D' para también mostrar el número de línea.

La función Mostrar_errores_ds_referencia

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función se encarga de mostrar por pantalla los errores específicos de las variables pasadas por referencia, es decir, comprueba que la DrCadena contenga al menos una 'D', puesto que una variable no tiene sentido que sea pasada por referencia si no es modificada al menos una vez en el interior de la rutina. Su funcionamiento es muy sencillo, tan solo comprueba carácter a carácter la cadena para ver si al menos hay una 'D', si no existiese ninguna mostraría el mensaje de error.

La función Mostrar_errores_rs_valor

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función se encarga de mostrar por pantalla los errores específicos de las variables pasadas por valor, es decir, comprueba que la DrCadena comience y termine por 'R', puesto que sería inútil modificar esta variable nada más entrar en la rutina y del mismo modo sería inútil que su DrCadena terminara con una modificación. Para ello se vale de las funciones 'PrimerElemento_Cadena' y 'UltimoElemento_Cadena', de este modo comprueba sus símbolos iniciales y finales. Si existiese el error mostraría un mensaje por pantalla.

La función Saca_NumLinea_error

- Entradas:
  - LineaVar: Variable tipo CLineaVar* usada para obtener la línea del código que se devuelve.

- NumS: Variable tipo *int* que indica la posición que ocupa el símbolo en la DrCadena.
- Symbol: Variable tipo *char* que indica que tipo de símbolo se ha de buscar, por lo tanto puede tener el valor 'R' o el valor 'D'.

- Salidas:
  - La función devuelve el numero de línea en la cual esta esa definición o esa referencia de la variable.

Funcionamiento

Esta función se encarga de obtener el número de línea en la cual se produce esa definición o esa referencia. La función tan solo cuenta uno a uno los símbolos facilitados mediante 'Symbol', ya sea 'R' o 'D', y una vez encuentra el que esta en la posición 'NumS' devuelve el número de línea en la que está.

La función SimbConsec_if

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - Pos: Variable tipo int que indica la posición que ocupa dentro de la DrCadena de 'DrCadena' el símbolo tratado, del que se desea obtener todos los símbolos que podrían ir inmediatamente después.
  - PosAux: Variable tipo *int* que indica el final de la estructura 'if' o 'case' cuyo comienzo se indica mediante la variable 'Pos'.
  - Lista_estruc: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos que están dentro de la estructura y que pueden ser consecutivos del que se esta tratando.
- Salidas:

- ListaSimb: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos pueden ser consecutivos del que se esta tratando.

Funcionamiento

Esta función se encarga de obtener los símbolos que van inmediatamente después de otro, en el caso de que este se encuentre dentro de una estructura 'if' o 'case'. Hace falta una rutina de estas características porque puede darse el caso de que una estructura 'if' esté dentro de una estructura iterativa y en ese caso símbolos consecutivos podrían ser todos los símbolos iniciales de cada una de las partes del 'if'.

La función SimbConsec_estructura

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - Pos: Variable tipo *int* que indica la posición que ocupa dentro de la DrCadena de 'DrCadena' el símbolo tratado, del que se desea obtener todos los símbolos que podrían ir inmediatamente después.
  - Lista_estruc: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos que están dentro de la estructura y que pueden ser consecutivos del que se esta tratando.
- Salidas:
  - ListaSimb: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos pueden ser consecutivos del que se esta tratando.

Funcionamiento

Esta función se encarga de obtener los símbolos que van inmediatamente después de otro, en el caso de que este se encuentre dentro de una estructura cualquiera. Primero identifica el tipo de estructura a tratar, en el caso de que sea una tipo 'if' llamaría a la rutina 'SimbConsec_if' y en caso contrario identificaría y almacenaría todos los símbolos que pudieran ser consecutivos, ya sean 'R' o 'D'.

La función Simbolos consecutivos

- Entradas:
  - DrCadena: Variable tipo *char**, esta cadena de caracteres se corresponde con una DrCadena.
  - Pos: Variable tipo *int* que indica la posición que ocupa dentro de la DrCadena de 'DrCadena' el símbolo tratado, del que se desea obtener todos los símbolos que podrían ir inmediatamente después.
  - Lista_estruc: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos que están dentro de la estructura y que pueden ser consecutivos del que se esta tratando.
- Salidas:
  - ListaSimb_siguen: Variable tipo *char** que almacena, separadas por comas, las posiciones de los símbolos pueden ser consecutivos del que se esta tratando.

Funcionamiento

Esta función es la encargada de devolver mediante una variable tipo *char** los símbolos que pueden ser consecutivos de otro concreto que tenemos identificado, es decir, habiendo recibido la posición de un símbolo de la cadena, ya puede ser 'D' o 'R', esta rutina almacena en una cadena de caracteres los símbolos que puede ir inmediatamente después. Estudiando las

estructuras, incluyendo en la que se encuentre la 'R' o 'D' especificada, se identifican los símbolos que puedan ir después teniendo en cuenta las estructuras iterativas y se almacenan tantas 'R' y tantas 'D' en la variable 'ListaSimb_siguen' como 'D' y 'R' puedan ir detrás.

La función Error_ds_consecutivas

- Entradas:
  - DrCadena Variable tipo *char**, esta cadena de caracteres se corresponde con la DrCadena que se va a analizar.
  - LineaVar: Variable tipo CLineaVar* usada para obtener la línea del código en la que esté una 'D' errónea.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función se encarga de mostrar por pantalla los errores tipo b) o de doble definición consecutiva que pueda tener una determinada cadena, la cual ha sido examinada previamente por la función 'Simbolos_consecutivos'. La función lee la cadena de principio a fin identificando todas las 'D' que haya, a cada una de ellas le aplica la rutina 'Simbolos_consecutivos' y obtiene almacenados en una cadena de caracteres todos los símbolos que pueden ser consecutivos de esa 'D'. Se estudia esa cadena y en caso de no haya ninguna 'R' y sin embargo si que haya 'D' se considera que la 'D' estudiada solo puede ser consecutiva de otra 'D' y por lo tanto es errónea. Se analiza la cadena completamente mostrando los mensajes de error por pantalla.

La función Errores_Dr_Cadena

- Entradas:
  - DrCadena Variable tipo *char**, esta cadena de caracteres se corresponde con la DrCadena que se va a analizar.

- LineaVar: Variable tipo CLineaVar* usada para obtener la línea del código en la que esté un error.
- Tipo: Variable tipo *char** que indica el tipo de variable ('local', 'global', 'referencia' o 'valor') cuya DrCadena se indica mediante la variable 'DrCadena'.

- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función es la más importante en lo que a análisis de las DrCadenas se refiere, es la encargada de comprobar que errores se producen en una DrCadena determinada. A la cadena recibida para analizar se le aplican las rutinas para detectar diferentes tipo de errores dependiendo de a que tipo de variable pertenezca, cosa que se indica mediante 'Tipo'. En la variables locales se busca si tienen los errores tipo a), b) y c) mediante las funciones definidas anteriormente 'Error_rs_iniciales', 'Error_ds_consecutivas' y 'Error_ds_finales' respectivamente y en caso afirmativo se muestran por pantalla los mensajes de error pertinentes. En las variables pasadas por referencia se busca si tiene el error tipo b) o de definiciones consecutivas usando la rutina 'Error_ds_consecutivas' y también se comprueba si tiene el error propio de los parámetros pasados por referencia, es decir que no exista ninguna 'D' en la cadena, mediante la función 'Mostrar_errores_ds_referencia'. Por útimo a la variables pasadas por valor se busca si tiene el error tipo b) o de definiciones consecutivas usando la rutina 'Error_ds_consecutivas' y también se comprueba si tiene el error propio de los parámetros pasados por valor, es decir que la cadena no comience ni empiece por 'R', mediante la función 'Mostrar_errores_rs_valor'. Las variables globales no se tratan puesto que luego se analizan debidamente en el programa principal.

La función Muestra

- Entradas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función es la muy importante, ya que se encarga de mostrar por pantalla las variables, su tipo, su DRCadena y sus errores en el caso de que los tenga. Esta rutina lee la estructura de datos almacenada en programa íntegra, comenzando en la primera variable de la primera función y acabando en la última variable de la última función. Para cada función que lee muestra en pantalla su nombre y todas sus variables una a una. Para cada variable que lee muestra por pantalla de forma ordenada su nombre, su tipo ('local', 'global', 'referencia' o 'valor') y su DrCadena, para posteriormente mostrar también los errores que tiene la DrCadena, en el caso de que los tenga, indicando para cada error su tipo, el símbolo en el cual sucede y la línea del programa en la que ocurre. Esto se aplica para todas las variables que tenga la estructura de datos.

La función Crea_Salida

- Entradas:
  - Programa: Objeto de la clase CPrograma en la cual se almacenan todos los datos concernientes a las variables que aparecen a lo largo del programa.
- Salidas:
  - No tiene parámetros de salida.

Funcionamiento

Esta función es la encargada de almacenar en un fichero de texto todos los datos contenidos en la estructura 'Programa' para su posterior tratamiento mediante otras aplicaciones. Devuelve en el fichero todos los datos obtenidos en el proceso del lectura, que no en el de análisis. Para ello muestra todos los datos contenidos en cada nodo de cada lista de la estructura. Comienza mostrando los datos del programa, luego para cada función muestra los datos propios de la función y los datos de cada una de sus variables, para cada variable muestra los datos propios de la variable y los datos de cada una de sus apariciones (objetos de la clase 'LineaVar'), y por último de cada aparición muestra sus datos, es decir, número de línea, posición en la línea y valor. Esto se hace para todas las listas que contiene la estructura de datos almacenada en 'Programa', desde el principio hasta el fin.

5.1.3.3 Función MAIN

He añadido este apartado porque creo que es necesario explicar aunque sea brevemente lo que hace esta importante función del programa, puesto que aunque he explicado el resto de funciones esta no la he tratado ya que no forma parte de la diferenciación echa anteriormente de Bloque1 y Bloque2, es decir, no pertenece ni a las funciones de lectura ni a las funciones de análisis, ya que obviamente este es la función encargada de llamar al Bloque1 y al Bloque2.

Su funcionamiento es extraordinariamente simple ya que tan solo llama a las rutinas 'LeerPrograma' (que se corresponde con el inicio del Bloque1) y 'Muestra' (que se corresponde con el inicio del Bloque2).

Pero aunque sencilla es vital, es por eso que la he incluido en un apartado propio y no en los anteriores.

5.2. MANUAL DE USUARIO

En este apartado se indicará al usuario como tratar con la aplicación de forma general.

El manejo de la aplicación es muy simple ya que la parte de lectura y análisis se realiza siempre de forma automática, el único modo de interactuar con el programa es mediante la entrada; la salida siempre se realiza del mismo modo.

La aplicación como entrada necesita un archivo de texto que contenga el código un programa compilado correctamente en el lenguaje de programación Pascal. El nombre de este archivo se pide nada más ejecutar la aplicación y después el programa comienza automáticamente los procesos de lectura del mismo y de análisis de la estructura de datos que genera.

Los resultados del análisis se muestran de forma ordenada por pantalla para que el usuario los entienda fácilmente.

Lo último que hace la aplicación es generar un archivo de texto el cual contiene de modo ordenado toda la estructura de datos generada para que lo vea el usuario y para un posible posterior tratamiento.

6. EXPERIMENTACIÓN

Se han estudiado diez ejemplos de experimentación que se presentan en los apartados siguientes. Se ha intentado que, en su conjunto, reflejen en su totalidad los diversos casos que se pueden dar. Los ocho primeros son casos sencillos que prueban una funcionalidad básica del programa, tal como lectura de estructuras simples. Los dos últimos son más complejos y engloban un mayor número de problemas.

Ejemplo 1: Estructura IF y ELSE.

En este primer ejemplo se verá como la aplicación trata una estructura *if*, tanto en su parte verdadera como en su parte falsa. También se mostrará la estructura de datos y el análisis de las DrCadenas.

```
 1. Program EjemploA1;
 2.
 3. VAR a,b,c,e:integer;
 4. arriba:boolean;
 5.
 6. BEGIN
 7. arriba:=true;
 8.       if arriba then begin
 9.  a:=1;
10.  b:=a;
11.       end
12.       else begin
13.  a:= c + 2;
14.  e:=1;
15.         arriba:= not (arriba);
16.       end;
17. e:=a;
18. END.
```

Como resultado de la lectura del código la estructura de datos nos queda como se indica a continuación:

```
PROGRAMA ejemploa1
        FUNCION main Linea_inicio 6 Linea_fin 18
                VARIABLE a Tipo local Posicion 0
```

Valor (NumLinea 8 PosLinea 2
Valor D NumLinea 9 PosLinea 1
Valor R NumLinea 10 PosLinea 1
Valor o NumLinea 12 PosLinea 1
Valor D NumLinea 13 PosLinea 2
Valor) NumLinea 17 PosLinea 1
Valor R NumLinea 17 PosLinea 2
VARIABLE b Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 2
Valor D NumLinea 10 PosLinea 2
Valor o NumLinea 12 PosLinea 1
Valor) NumLinea 17 PosLinea 1
VARIABLE c Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 12 PosLinea 1
Valor R NumLinea 13 PosLinea 1
Valor) NumLinea 17 PosLinea 1
VARIABLE e Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 12 PosLinea 1
Valor D NumLinea 14 PosLinea 1
Valor) NumLinea 17 PosLinea 1
Valor D NumLinea 17 PosLinea 3
VARIABLE arriba Tipo local Posicion 0
Valor D NumLinea 7 PosLinea 1
Valor R NumLinea 8 PosLinea 1
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 12 PosLinea 1
Valor R NumLinea 15 PosLinea 1
Valor D NumLinea 15 PosLinea 2
Valor) NumLinea 17 PosLinea 1

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable, ya que en este caso la lista de funciones se compone tan solo de un nodo, el del programa principal (llamado MAIN).

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

    DrCadena: (DRoD)R

Errores:

- Ninguno

- Variable : b Tipo : Local

  DrCadena: (**D**o1)

  Errores:

  - Error por definición final en variable local, error en la D número 1 (línea 10)

- Variable : c Tipo : Local

  DrCadena: (1o**R**)

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 13)

- Variable : e Tipo : Local

  DrCadena: (1o**D**)**D**

  Errores:

  - Error por definición final en variable local, error en la D número 2 (línea 17)
  - Error por definición consecutiva en variable local, error en la D número 1 (línea 14)

- Variable : arriba Tipo : Local

  DrCadena: DR(1oR**D**)

  Errores:

  - Error por definición final en variable local, error en la D número 2 (línea 15)

Ejemplo 2: Estructura CASE y ELSE.

En este ejemplo se verá como la aplicación trata una estructura *case*, tanto en sus partes verdaderas como en su parte falsa. También se mostrará la estructura de datos y el análisis de las DrCadenas.

```
 1. Program EjemploA2;
 2.
 3. VAR a,b,c:integer;
 4.   ch:char;
 5.
 6. BEGIN
 7. a:=b;
 8. case ch of
 9. 'A'..'Z':begin
10.       a:=1;
11.       b:=a;
12.    end;
13. '0'..'9':begin
14.       a:=2;
15.       c:=a;
16.       a:=c;
17.    end;
18. '','':a:=3
19. else begin
20.   a:=4;
21.   d:=a;
22.       end;
23. END.
```

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

PROGRAMA ejemploa2
FUNCION main Linea_inicio 6 Linea_fin 23
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 7 PosLinea 2
Valor (NumLinea 9 PosLinea 1
Valor D NumLinea 10 PosLinea 1
Valor R NumLinea 11 PosLinea 1
Valor o NumLinea 13 PosLinea 1
Valor D NumLinea 14 PosLinea 1
Valor R NumLinea 15 PosLinea 1

Valor D NumLinea 16 PosLinea 2
Valor o NumLinea 18 PosLinea 1
Valor D NumLinea 20 PosLinea 3
Valor o NumLinea 21 PosLinea 1
Valor o NumLinea 22 PosLinea 1
Valor) NumLinea 23 PosLinea 1
VARIABLE b Tipo local Posicion 0
Valor R NumLinea 7 PosLinea 1
Valor (NumLinea 9 PosLinea 1
Valor D NumLinea 11 PosLinea 2
Valor o NumLinea 13 PosLinea 1
Valor o NumLinea 18 PosLinea 1
Valor o NumLinea 21 PosLinea 1
Valor o NumLinea 22 PosLinea 1
Valor) NumLinea 23 PosLinea 1
VARIABLE c Tipo local Posicion 0
Valor (NumLinea 9 PosLinea 1
Valor o NumLinea 13 PosLinea 1
Valor D NumLinea 15 PosLinea 2
Valor R NumLinea 16 PosLinea 1
Valor o NumLinea 18 PosLinea 1
Valor o NumLinea 21 PosLinea 1
Valor o NumLinea 22 PosLinea 1
Valor) NumLinea 23 PosLinea 1
VARIABLE ch Tipo local Posicion 0
Valor R NumLinea 8 PosLinea 1
Valor (NumLinea 9 PosLinea 1
Valor o NumLinea 13 PosLinea 1
Valor o NumLinea 18 PosLinea 1
Valor o NumLinea 21 PosLinea 1
Valor o NumLinea 22 PosLinea 1
Valor) NumLinea 23 PosLinea 1

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable, ya que en este caso la lista de funciones se compone tan solo de un nodo, el del programa principal (llamado MAIN).

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

DrCadena: **D**(DRoDR**D**o**D**o1o1)

Errores:

- Error por definición final en variable local, error en la D número 5 (línea 20)
- Error por definición final en variable local, error en la D número 4 (línea 16)
- Error por definición consecutiva en variable local, error en la D número 1 (línea 7)

- Variable : b Tipo : Local

  DrCadena: **R**(**D**o1o1o1o1)

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 7)
  - Error por definición final en variable local, error en la D número 1 (línea 11)

- Variable : c Tipo : Local

  DrCadena: (1oDRo1o1o1)

  Errores:

  - No hay errores

- Variable : ch Tipo : Local

  DrCadena: R

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 8)

Ejemplo 3: Estructura WHILE.

En este ejemplo se verá como la aplicación trata una estructura *while*, cuyo análisis es un poco diferente dado que es una estructura iterativa. También se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA3;
 2.
 3. VAR a,b,c,d:integer;
 4.
 5. BEGIN
 6. a:=10;
 7. while (a<20) do
 8.    begin
 9.        if (a=15) then b:=10;
10.        b:=c;
11.        c:=15;
12.        read(d);
13.        inc(a);
14.    end;
15. END.
```

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

PROGRAMA ejemploa3
FUNCION main Linea_inicio 5 Linea_fin 15
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 6 PosLinea 1
Valor R NumLinea 7 PosLinea 1
Valor (NumLinea 8 PosLinea 1
Valor R NumLinea 9 PosLinea 1
Valor (NumLinea 9 PosLinea 2
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor D NumLinea 13 PosLinea 1
Valor R NumLinea 15 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE b Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 1
Valor (NumLinea 9 PosLinea 2

Valor D NumLinea 9 PosLinea 3
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor D NumLinea 10 PosLinea 4
Valor)* NumLinea 15 PosLinea 2
VARIABLE c Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 1
Valor (NumLinea 9 PosLinea 2
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor R NumLinea 10 PosLinea 3
Valor D NumLinea 11 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE d Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 1
Valor (NumLinea 9 PosLinea 2
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor D NumLinea 12 PosLinea 1
Valor)* NumLinea 15 PosLinea 2

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable, ya que en este caso la lista de funciones se compone tan solo de un nodo, el del programa principal (llamado MAIN).

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

    DrCadena: DR(RDR)*

    Errores:
    - No hay errores
  - Variable : b Tipo : Local

    DrCadena: ((**D**o1)**D**)*

    Errores:
    - Error por definición final en variable local, error en la D número 2 (línea 10)

- Error por definición consecutiva en variable local, error en la D número 1 (línea 9)

- Variable : c Tipo : Local

  DrCadena: (**RD**)*

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 10)
  - Error por definición final en variable local, error en la D número 1 (línea 11)

- Variable : d Tipo : Local

  DrCadena: **D***

  Errores:

  - Error por definición final en variable local, error en la D número 1 (línea 12)

Ejemplo 4: Estructura REPEAT.

En este ejemplo se verá como la aplicación trata una estructura *repeat*, cuyo análisis es un poco diferente dado que es una estructura iterativa. También se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA4;
 2.
 3. VAR a,b,c:integer;
 4.
 5. BEGIN
 6. a:=1;
 7. repeat
 8.    b:=c;
 9.    if (b=2) then b:=15;
10.    a:=a+1;
11. until (a=10);
12. END.
```

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

PROGRAMA ejemploa4
FUNCION main Linea_inicio 5 Linea_fin 12
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 6 PosLinea 1
Valor (NumLinea 8 PosLinea 1
Valor (NumLinea 9 PosLinea 2
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor R NumLinea 10 PosLinea 3
Valor D NumLinea 10 PosLinea 4
Valor R NumLinea 11 PosLinea 1
Valor)+ NumLinea 11 PosLinea 2
VARIABLE b Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 1
Valor D NumLinea 8 PosLinea 3
Valor R NumLinea 9 PosLinea 1
Valor (NumLinea 9 PosLinea 2
Valor D NumLinea 9 PosLinea 3

Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor)+ NumLinea 11 PosLinea 2
VARIABLE c Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 1
Valor R NumLinea 8 PosLinea 2
Valor (NumLinea 9 PosLinea 2
Valor o NumLinea 10 PosLinea 1
Valor) NumLinea 10 PosLinea 2
Valor)+ NumLinea 11 PosLinea 2

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable, ya que en este caso la lista de funciones se compone tan solo de un nodo, el del programa principal (llamado MAIN).

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local
    DrCadena: D(RDR)+
    Errores:
    - No hay errores
  - Variable : b Tipo : Local
    DrCadena: (DR(**D**o1))+
    Errores:
    - Error por definición final en variable local, error en la D número 2 (línea 9)
  - Variable : c Tipo : Local
    DrCadena: **R**+
    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 8)

Ejemplo 5: Estructura FOR.

En este ejemplo se verá como la aplicación trata una estructura *for*, cuyo análisis es un poco diferente dado que es una estructura iterativa, además para tratar este tipo de estructuras se aplica un algoritmo que convierte el *for* en una estructura *while* equivalente. También se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA5;
 2.
 3. VAR a,b,c,d:integer;
 4.
 5. BEGIN
 6. for a:=1 to 10 do
 7.    begin
 8.       if (a=5) then (b:=c) else (b:=1);
 9.       c:=a + b;
10.       if (c=a) then d:=a;
11.    end;
12. END.
```

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

PROGRAMA ejemploa5
FUNCION main Linea_inicio 5 Linea_fin 12
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 6 PosLinea 1
Valor R NumLinea 6 PosLinea 2
Valor (NumLinea 6 PosLinea 3
Valor R NumLinea 8 PosLinea 1
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 8 PosLinea 5
Valor) NumLinea 9 PosLinea 1
Valor R NumLinea 9 PosLinea 2
Valor R NumLinea 10 PosLinea 2
Valor (NumLinea 10 PosLinea 3
Valor R NumLinea 10 PosLinea 4
Valor o NumLinea 11 PosLinea 1

```
        Valor ) NumLinea 11 PosLinea 2
        Valor RDR NumLinea 12 PosLinea 1
        Valor )n NumLinea 12 PosLinea 2
VARIABLE b Tipo local Posicion 0
        Valor ( NumLinea 6 PosLinea 3
        Valor ( NumLinea 8 PosLinea 2
        Valor o NumLinea 8 PosLinea 5
        Valor ) NumLinea 9 PosLinea 1
        Valor R NumLinea 9 PosLinea 3
        Valor ( NumLinea 10 PosLinea 3
        Valor o NumLinea 11 PosLinea 1
        Valor ) NumLinea 11 PosLinea 2
        Valor )n NumLinea 12 PosLinea 2
VARIABLE c Tipo local Posicion 0
        Valor ( NumLinea 6 PosLinea 3
        Valor ( NumLinea 8 PosLinea 2
        Valor R NumLinea 8 PosLinea 3
        Valor o NumLinea 8 PosLinea 5
        Valor ) NumLinea 9 PosLinea 1
        Valor D NumLinea 9 PosLinea 4
        Valor R NumLinea 10 PosLinea 1
        Valor ( NumLinea 10 PosLinea 3
        Valor o NumLinea 11 PosLinea 1
        Valor ) NumLinea 11 PosLinea 2
        Valor )n NumLinea 12 PosLinea 2
VARIABLE d Tipo local Posicion 0
        Valor ( NumLinea 6 PosLinea 3
        Valor ( NumLinea 8 PosLinea 2
        Valor o NumLinea 8 PosLinea 5
        Valor ) NumLinea 9 PosLinea 1
        Valor ( NumLinea 10 PosLinea 3
        Valor D NumLinea 10 PosLinea 5
        Valor o NumLinea 11 PosLinea 1
        Valor ) NumLinea 11 PosLinea 2
        Valor )n NumLinea 12 PosLinea 2
```

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable, ya que en este caso la lista de funciones se compone tan solo de un nodo, el del programa principal (llamado MAIN).

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

    DrCadena: DR(RRR(Ro1)RDR)n

    Errores:
    - No hay errores
  - Variable : b Tipo : Local

    DrCadena: **R**n

    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 9)
  - Variable : c Tipo : Local

    DrCadena: ((**R**o1)DR)n

    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 8)
  - Variable : d Tipo : Local

    DrCadena: (**D**o1)n

    Errores:
    - Error por definición final en variable local, error en la D número 1 (línea 10)
    - Error por definición consecutiva en variable local, error en la D número 1 (línea 10)

Ejemplo 6: Parámetros por valor y por referencia.

En este ejemplo se verá como la aplicación trata una llamada a una función en cual hay variables pasadas por valor y variables pasadas por referencia, y como esto influye en la DrCadena de estas variables. También se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA6;
 2.
 3. VAR d,e,f:integer;
 4.
 5. Procedure Rutina1(Var a:int; b:int);
 6. Var c:int;
 7. Begin
 8.   b:=a;
 9.   c:=b;
10.   if (c<5) then c:=a;
11. End;
12.
13. BEGIN
14.    d:=1;
15.    e:=d;
16.    f:=e;
17.    Rutina1(d,e);
18. END.
```

La variable ‘d’ esta pasada por referencia por lo que a su DrCadena se le añade la cadena generada por la variable ‘a’ dentro del procedimiento ‘Rutina1’.

La variable ‘e’ esta pasada por valor por lo que a su DrCadena se le debe añadir una ‘R’ cuando se llama al procedimiento ‘Rutina1’.

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

```
PROGRAMA ejemploa6
        FUNCION main Linea_inicio 13 Linea_fin 18
                VARIABLE d Tipo local Posicion 0
                        Valor D NumLinea 14 PosLinea 1
```

```
                Valor R NumLinea 15 PosLinea 1
                Valor R(Ro) NumLinea 17 PosLinea 2
        VARIABLE e Tipo local Posicion 0
                Valor D NumLinea 15 PosLinea 2
                Valor R NumLinea 16 PosLinea 1
                Valor R NumLinea 17 PosLinea 1
        VARIABLE f Tipo local Posicion 0
                Valor D NumLinea 16 PosLinea 2
FUNCION rutina1 Linea_inicio 5 Linea_fin 11
        VARIABLE a Tipo referencia Posicion 1
                Valor R NumLinea 8 PosLinea 1
                Valor ( NumLinea 10 PosLinea 2
                Valor R NumLinea 10 PosLinea 3
                Valor o NumLinea 11 PosLinea 1
                Valor ) NumLinea 11 PosLinea 2
        VARIABLE b Tipo valor Posicion 2
                Valor D NumLinea 8 PosLinea 2
                Valor R NumLinea 9 PosLinea 1
                Valor ( NumLinea 10 PosLinea 2
                Valor o NumLinea 11 PosLinea 1
                Valor ) NumLinea 11 PosLinea 2
        VARIABLE c Tipo local Posicion 0
                Valor D NumLinea 9 PosLinea 2
                Valor R NumLinea 10 PosLinea 1
                Valor ( NumLinea 10 PosLinea 2
                Valor D NumLinea 10 PosLinea 4
                Valor o NumLinea 11 PosLinea 1
                Valor ) NumLinea 11 PosLinea 2
```

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable; en este caso la lista de funciones se compone de dos nodos, el del programa principal (llamado MAIN) y el de la rutina usada.

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : d Tipo : Local

    DrCadena: DRR(Ro1)

    Errores:
    - No hay errores

- Variable : e Tipo : Local

  DrCadena: DRR

  Errores:

  - No hay errores

- Variable : f Tipo : Local

  DrCadena: **D**

  Errores:

  - Error por definición final en variable local, error en la D número 1 (línea 16)

♦ Función: Rutina1

- Variable : a Tipo : Referencia

  DrCadena: R(Ro1)

  Errores:

  - Error por no existencia de definición en variable pasada por referencia

- Variable : b Tipo : Valor

  DrCadena: DR

  Errores:

  - Error por no existencia de referencia inicial o final en variable pasada por valor

- Variable : c Tipo : Local

  DrCadena: DR(**D**o1)

  Errores:

  - Error por definición final en variable local, error en la D número 2 (línea 10)

Ejemplo 7: Función con globales y parámetros por valor y por referencia.

En este ejemplo se verá como la aplicación trata una llamada a una función en cual hay variables pasadas por valor, variables pasadas por referencia, y variables globales. Se verá una variable que tan solo esta como global y otras que además de estar pasadas como parámetros también están como globales. A continuación se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA7;
 2.
 3. VAR f,g,h,i:integer;
 4.
 5. Procedure Rutina1(Var a,b:int; c,d:int);
 6. Var e:int;
 7. Begin
 8.    if (b>1) then a:=a+1;
 9.    f:=b;
11.    while (d<=5) do
12.       begin
13.          c:=5;
14.          h:=d;
15.       end;
16. End;
17.
18. BEGIN
19.    f:=0;
20.    g:=0;
21.    h:=0;
22.    i:=0;
23.    Rutina1(f,g,h,i);
24. END.
```

La variable ‘g’ esta pasada por referencia por lo que a su DrCadena se le añade la cadena generada por la variable ‘b’ dentro del procedimiento ‘Rutina1’.

La variable ‘f’ esta pasada por referencia y además está como variable global, por lo que a su DrCadena se le añade la unión de la cadena generada por la variable ‘a’ dentro del procedimiento ‘Rutina1’ y la cadena de ‘f’ dentro de ‘Rutina1’.

La variable ‘i’ esta pasada por valor por lo que a su DrCadena se le debe añadir una ‘R’ cuando se llama al procedimiento ‘Rutina1’.

La variable 'h' esta pasada por valor y además está como variable global, por lo que su DrCadena se compone de la cadena de 'h' dentro de 'Rutina1' mas la 'R' que indica que ha sido pasada por valor.

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

```
PROGRAMA ejemploa7
        FUNCION main Linea_inicio 17 Linea_fin 23
                VARIABLE f Tipo local Posicion 0
                        Valor D NumLinea 18 PosLinea 1
                        Valor (RDo)D()* NumLinea 22 PosLinea 3
                VARIABLE g Tipo local Posicion 0
                        Valor D NumLinea 19 PosLinea 1
                        Valor R(o)R()* NumLinea 22 PosLinea 4
                VARIABLE h Tipo local Posicion 0
                        Valor D NumLinea 20 PosLinea 1
                        Valor R NumLinea 22 PosLinea 1
                        Valor (o)(D)* NumLinea 22 PosLinea 5
                VARIABLE i Tipo local Posicion 0
                        Valor D NumLinea 21 PosLinea 1
                        Valor R NumLinea 22 PosLinea 2
        FUNCION rutina1 Linea_inicio 5 Linea_fin 15
                VARIABLE a Tipo referencia Posicion 1
                        Valor ( NumLinea 8 PosLinea 2
                        Valor R NumLinea 8 PosLinea 3
                        Valor D NumLinea 8 PosLinea 4
                        Valor o NumLinea 9 PosLinea 1
                        Valor ) NumLinea 9 PosLinea 2
                        Valor ( NumLinea 11 PosLinea 1
                        Valor)* NumLinea 15 PosLinea 2
                VARIABLE b Tipo referencia Posicion 2
                        Valor R NumLinea 8 PosLinea 1
                        Valor ( NumLinea 8 PosLinea 2
                        Valor o NumLinea 9 PosLinea 1
                        Valor ) NumLinea 9 PosLinea 2
                        Valor R NumLinea 9 PosLinea 3
                        Valor ( NumLinea 11 PosLinea 1
                        Valor )* NumLinea 15 PosLinea 2
                VARIABLE c Tipo valor Posicion 3
                        Valor ( NumLinea 8 PosLinea 2
                        Valor o NumLinea 9 PosLinea 1
```

Valor) NumLinea 9 PosLinea 2
Valor (NumLinea 11 PosLinea 1
Valor D NumLinea 12 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE d Tipo valor Posicion 4
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 9 PosLinea 1
Valor) NumLinea 9 PosLinea 2
Valor R NumLinea 10 PosLinea 1
Valor (NumLinea 11 PosLinea 1
Valor R NumLinea 13 PosLinea 1
Valor R NumLinea 15 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE e Tipo local Posicion 0
Valor (NumLinea 8 PosLinea 2
Valor o NumLinea 9 PosLinea 1
Valor) NumLinea 9 PosLinea 2
Valor (NumLinea 11 PosLinea 1
Valor)* NumLinea 15 PosLinea 2

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable; en este caso la lista de funciones se compone de dos nodos, el del programa principal (llamado MAIN) y el de la rutina usada.

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : f Tipo : Local

    DrCadena: D(R**D**o1)**D**

    Errores:
    - Error por definición final en variable local, error en la D número 3 (línea 22)
    - Error por definición consecutiva en variable local, error en la D número 2 (línea 22)
  - Variable : g Tipo : Local

    DrCadena: DRR

Errores:

- No hay errores

- Variable : h Tipo : Local

  DrCadena: DR**D***

  Errores:

  - Error por definición final en variable local, error en la D número 2 (línea 22)

- Variable : i Tipo : Local

  DrCadena: DR

  Errores:

  - No hay errores

♦ Función: Rutina1

- Variable : a Tipo : Referencia

  DrCadena: (RDo1)

  Errores:

  - No hay errores

- Variable : b Tipo : Referencia

  DrCadena: RR

  Errores:

  - Error por no existencia de definición en variable pasada por referencia

- Variable : c Tipo : Valor

  DrCadena: D*

  Errores:

  - Error por no existencia de referencia inicial o final en variable pasada por valor

- Variable : d Tipo : Valor

  DrCadena: R(RR)*

  Errores:

  - No hay errores

- Variable : e Tipo : Local

DrCadena:

Errores:

- Error por no utilización de la variable

- Variable : f Tipo : Global

  DrCadena: D

  Errores:

  - No se analizan las globales

- Variable : h Tipo : Global

  DrCadena: D*

  Errores:

  - No se analizan las globales

Ejemplo 8: Función con el mismo parámetro en diversas posiciones.

En este ejemplo se verá como la aplicación trata una llamada a una función en cual hay variables como parámetros en diversas posiciones. Se verá una variable que esta pasada a la vez como valor y referencia, otra que esta como referencia dos veces, etc... combinándolo con variables locales y globales. Después se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploA8;
 2.
 3. VAR f,g,h:integer;
 4.
 5. Procedure Rutina1(Var a,b:int; VAR c:int; d,e:int);
 6. Var e:int;
 7. Begin
 8.    if (e>1) then a:=a+1;
 9.    f:=b;
10.    while (d<=5) do
11.       begin
12.          c:=5;
13.          c:=c+1;
14.          d:=c;
15.       end;
16. End;
17.
18. BEGIN
19.    f:=0;
20.    g:=0;
21.    h:=0;
22.    Rutina1(g,g,h,h,f);
23. END.
```

La variable ‘g’ esta pasada por referencia dos veces, es decir en dos posiciones, por lo que a su DrCadena se le añade las cadenas generadas por las variables ‘a’ y ‘b’ dentro del procedimiento ‘Rutina1’.

La variable ‘f’ esta pasada por valor y además está como variable global, por lo que su DrCadena se compone de la cadena de ‘f’ dentro de ‘Rutina1’ mas la ‘R’ que indica que ha sido pasada por valor.

La variable ‘h’ esta pasada por valor y por referencia, es decir en dos posiciones, por lo que a su DrCadena se le añade la unión de la cadena generada por

la variable ‘c’ dentro del procedimiento ‘Rutina1’ mas la ‘R’ que indica que ha sido pasada por valor.

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

```
PROGRAMA ejemploa8
        FUNCION main Linea_inicio 17 Linea_fin 22
                VARIABLE f Tipo local Posicion 0
                        Valor D NumLinea 18 PosLinea 1
                        Valor R NumLinea 21 PosLinea 2
                        Valor (o)D()* NumLinea 21 PosLinea 5
                VARIABLE g Tipo local Posicion 0
                        Valor D NumLinea 19 PosLinea 1
                        Valor (RDo)R()* NumLinea 21 PosLinea 3
                VARIABLE h Tipo local Posicion 0
                        Valor D NumLinea 20 PosLinea 1
                        Valor R NumLinea 21 PosLinea 1
                        Valor (o)(DRDR)* NumLinea 21 PosLinea 4
        FUNCION rutina1 Linea_inicio 5 Linea_fin 15
                VARIABLE a Tipo referencia Posicion 1
                        Valor ( NumLinea 7 PosLinea 2
                        Valor R NumLinea 7 PosLinea 3
                        Valor D NumLinea 7 PosLinea 4
                        Valor o NumLinea 8 PosLinea 1
                        Valor ) NumLinea 8 PosLinea 2
                        Valor ( NumLinea 10 PosLinea 1
                        Valor )* NumLinea 15 PosLinea 2
                VARIABLE b Tipo referencia Posicion 2
                        Valor ( NumLinea 7 PosLinea 2
                        Valor o NumLinea 8 PosLinea 1
                        Valor ) NumLinea 8 PosLinea 2
                        Valor R NumLinea 8 PosLinea 3
                        Valor ( NumLinea 10 PosLinea 1
                        Valor )* NumLinea 15 PosLinea 2
                VARIABLE c Tipo referencia Posicion 3
                        Valor ( NumLinea 7 PosLinea 2
                        Valor o NumLinea 8 PosLinea 1
                        Valor ) NumLinea 8 PosLinea 2
                        Valor ( NumLinea 10 PosLinea 1
                        Valor D NumLinea 11 PosLinea 1
                        Valor R NumLinea 12 PosLinea 1
```

Valor D NumLinea 12 PosLinea 2
Valor R NumLinea 13 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE d Tipo valor Posicion 4
Valor (NumLinea 7 PosLinea 2
Valor o NumLinea 8 PosLinea 1
Valor) NumLinea 8 PosLinea 2
Valor R NumLinea 9 PosLinea 1
Valor (NumLinea 10 PosLinea 1
Valor D NumLinea 13 PosLinea 2
Valor R NumLinea 15 PosLinea 1
Valor)* NumLinea 15 PosLinea 2
VARIABLE e Tipo valor Posicion 5
Valor R NumLinea 7 PosLinea 1
Valor (NumLinea 7 PosLinea 2
Valor o NumLinea 8 PosLinea 1
Valor) NumLinea 8 PosLinea 2
Valor (NumLinea 10 PosLinea 1
Valor)* NumLinea 15 PosLinea 2

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable; en este caso la lista de funciones se compone de dos nodos, el del programa principal (llamado MAIN) y el de la rutina usada.

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : f Tipo : Local

    DrCadena: DR**D**

    Errores:
    - Error por definición final en variable local, error en la D número 2 (línea 21)
  - Variable : g Tipo : Local

    DrCadena: D(RDo1)R

    Errores:
    - No hay errores
  - Variable : h Tipo : Local

DrCadena: DR(DRDR)*

Errores:

- No hay errores

- Función: Rutina1
  - Variable : a Tipo : Referencia

    DrCadena: (RDo1)

    Errores:

    - No hay errores
  - Variable : b Tipo : Referencia

    DrCadena: R

    Errores:

    - Error por no existencia de definición en variable pasada por referencia
  - Variable : c Tipo : Referencia

    DrCadena: (DRDR)*

    Errores:

    - No hay errores
  - Variable : d Tipo : Valor

    DrCadena: R(DR)*

    Errores:

    - No hay errores
  - Variable : e Tipo : Valor

    DrCadena: R

    Errores:

    - No hay errores
  - Variable : f Tipo : Global

    DrCadena: D

    Errores:

    - No se analizan las globales

Ejemplo 9: Primer caso de código más complejo con diversas estructuras y funciones.

En este ejemplo se verá como la aplicación trata un programa en el cual hay definidas dos rutinas, con diversos tipos de estructuras cada una, y en cuyo código principal se llama a estas en diversas ocasiones y dentro de diferentes estructuras. Como se comprobará se trata de un caso más complejo que los anteriores y mediante el cual se verá como funciona la estructura de datos de forma más completa. Después se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploB1;
 2.
 3. Var a,b,c,d:integer;
 4.
 5. Procedure Fastidio (Var x,y:integer;z:integer);
 6.   Var a,b:integer;
 7.   Begin
 8.     x:=0;
 9.     read(a,y);
10.     if (a>y) then d:=z
11.   End;
12.
13. Procedure Problema(a:integer;Var b:integer);
14.   Var c,d:integer;
15.   Begin
16.     c:= a+b;
17.     d:=c-b;
18.     repeat a:=b until (a>5)
19.   End;
20.
21. BEGIN
22.    b:=3;
23.    fastidio(b,c,b);
24.    if (c>0) then b:=d
25.       else c:=a;
26.    while (b>0)do
27.    begin
28.       problema(b,a);
29.       fastidio(c,a,b)
30.    end;
31.    writeln('la solucion es',a,b,c)
32. END.
```

Como se ve existen varios de los casos vistos anteriormente, ya que hay estructuras de diferentes tipos, rutinas con parámetros por valor y por referencia, variables globales (variable 'd' en procedimiento 'Fastidio'), variables pasadas como parámetros en diversas posiciones en la misma llamada a una función (variable 'b' en la primera llamada a procedimiento 'Fastidio'), y todo ello combinado, lo que es un problema añadido pero que el programa resuelve perfectamente.

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

```
PROGRAMA ejemplob1
        FUNCION main Linea_inicio 21 Linea_fin 32
                VARIABLE a Tipo local Posicion 0
                        Valor ( NumLinea 24 PosLinea 2
                        Valor o NumLinea 25 PosLinea 3
                        Valor R NumLinea 25 PosLinea 4
                        Valor ) NumLinea 26 PosLinea 1
                        Valor ( NumLinea 27 PosLinea 1
                        Valor RR(R)+ NumLinea 28 PosLinea 2
                        Valor DR(o) NumLinea 29 PosLinea 4
                        Valor )* NumLinea 31 PosLinea 2
                        Valor R NumLinea 31 PosLinea 3
                VARIABLE b Tipo local Posicion 0
                        Valor D NumLinea 22 PosLinea 1
                        Valor R NumLinea 23 PosLinea 2
                        Valor D(o) NumLinea 23 PosLinea 3
                        Valor ( NumLinea 24 PosLinea 2
                        Valor D NumLinea 25 PosLinea 2
                        Valor o NumLinea 25 PosLinea 3
                        Valor ) NumLinea 26 PosLinea 1
                        Valor R NumLinea 26 PosLinea 2
                        Valor ( NumLinea 27 PosLinea 1
                        Valor R NumLinea 28 PosLinea 1
                        Valor R NumLinea 29 PosLinea 2
                        Valor R NumLinea 31 PosLinea 1
                        Valor )* NumLinea 31 PosLinea 2
                        Valor R NumLinea 31 PosLinea 4
                VARIABLE c Tipo local Posicion 0
                        Valor DR(o) NumLinea 23 PosLinea 4
                        Valor R NumLinea 24 PosLinea 1
```

Valor (NumLinea 24 PosLinea 2
Valor o NumLinea 25 PosLinea 3
Valor D NumLinea 25 PosLinea 5
Valor) NumLinea 26 PosLinea 1
Valor (NumLinea 27 PosLinea 1
Valor D(o) NumLinea 29 PosLinea 3
Valor)* NumLinea 31 PosLinea 2
Valor R NumLinea 31 PosLinea 5
VARIABLE d Tipo local Posicion 0
Valor (Do) NumLinea 23 PosLinea 1
Valor (NumLinea 24 PosLinea 2
Valor R NumLinea 25 PosLinea 1
Valor o NumLinea 25 PosLinea 3
Valor) NumLinea 26 PosLinea 1
Valor (NumLinea 27 PosLinea 1
Valor (Do) NumLinea 29 PosLinea 1
Valor)* NumLinea 31 PosLinea 2
FUNCION fastidio Linea_inicio 5 Linea_fin 11
VARIABLE x Tipo referencia Posicion 1
Valor D NumLinea 8 PosLinea 1
Valor (NumLinea 10 PosLinea 3
Valor o NumLinea 11 PosLinea 3
Valor) NumLinea 11 PosLinea 4
VARIABLE y Tipo referencia Posicion 2
Valor D NumLinea 9 PosLinea 2
Valor R NumLinea 10 PosLinea 2
Valor (NumLinea 10 PosLinea 3
Valor o NumLinea 11 PosLinea 3
Valor) NumLinea 11 PosLinea 4
VARIABLE z Tipo valor Posicion 3
Valor (NumLinea 10 PosLinea 3
Valor R NumLinea 11 PosLinea 1
Valor o NumLinea 11 PosLinea 3
Valor) NumLinea 11 PosLinea 4
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 9 PosLinea 1
Valor R NumLinea 10 PosLinea 1
Valor (NumLinea 10 PosLinea 3
Valor o NumLinea 11 PosLinea 3
Valor) NumLinea 11 PosLinea 4
VARIABLE b Tipo local Posicion 0
Valor (NumLinea 10 PosLinea 3
Valor o NumLinea 11 PosLinea 3
Valor) NumLinea 11 PosLinea 4
FUNCION problema Linea_inicio 13 Linea_fin 19
VARIABLE a Tipo valor Posicion 1
Valor R NumLinea 16 PosLinea 1
Valor (NumLinea 18 PosLinea 1

Valor D NumLinea 18 PosLinea 3
Valor R NumLinea 18 PosLinea 4
Valor)+ NumLinea 19 PosLinea 1
VARIABLE b Tipo referencia Posicion 2
Valor R NumLinea 16 PosLinea 2
Valor R NumLinea 17 PosLinea 2
Valor (NumLinea 18 PosLinea 1
Valor R NumLinea 18 PosLinea 2
Valor)+ NumLinea 19 PosLinea 1
VARIABLE c Tipo local Posicion 0
Valor D NumLinea 16 PosLinea 3
Valor R NumLinea 17 PosLinea 1
Valor (NumLinea 18 PosLinea 1
Valor)+ NumLinea 19 PosLinea 1
VARIABLE d Tipo local Posicion 0
Valor D NumLinea 17 PosLinea 3
Valor (NumLinea 18 PosLinea 1
Valor)+ NumLinea 19 PosLinea 1

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable; en este caso la lista de funciones se compone de tres nodos, el del programa principal (llamado MAIN) y los de las dos rutinas usadas ('Fastidio' y 'Problema').

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

    DrCadena: (1o**R**)(RRR+DR)*R

    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 25)
  - Variable : b Tipo : Local

    DrCadena: DRD(Do1)R(RRR)*R

    Errores:
    - No hay errores

- Variable : c Tipo : Local

  DrCadena: DRR(1o**D**)D*R

  Errores:

  - Error por definición consecutiva en variable local, error en la D número 2 (línea 25)

- Variable : d Tipo : Local

  DrCadena: (Do1)(Ro1)((**D**o1))*

  Errores:

  - Error por definición final en variable local, error en la D número 2 (línea 29)

♦ Función: Fastidio

- Variable : x Tipo : Referencia

  DrCadena: D

  Errores:

  - No hay errores

- Variable : y Tipo : Referencia

  DrCadena: DR

  Errores:

  - No hay errores

- Variable : z Tipo : Valor

  DrCadena: (Ro1)

  Errores:

  - No hay errores

- Variable : a Tipo : Local

  DrCadena: DR

  Errores:

  - No hay errores

- Variable : b Tipo : Local

  DrCadena:

  Errores:

  - Error por no utilización de la variable

- Variable : d Tipo : Global

  DrCadena: (Do1)

  Errores:

  - No se analizan las globales

♦ Función: Problema

- Variable : a Tipo : Valor

  DrCadena: R(DR)+

  Errores:

  - No hay errores

- Variable : b Tipo : Referencia

  DrCadena: RRR+

  Errores:

  - Error por no existencia de definición en variable pasada por referencia

- Variable : c Tipo : Local

  DrCadena: DR

  Errores:

  - No hay errores

- Variable : d Tipo : Local

  DrCadena: **D**

  Errores:

  - Error por definición final en variable local, error en la D número 1 (línea 17)

Ejemplo 10: Segundo caso de código más complejo con diversas estructuras y funciones.

Como en el ejemplo anterior en este se verá un caso más complejo, se comprobará como la aplicación trata un programa en el cual hay definidas tres rutinas, con diversos tipos de estructuras cada una, y en cuyo código principal se llama a estas en diversas ocasiones y dentro de diferentes estructuras. También se mostrará el caso de que dentro de una rutina se llame a otra, caso no tratado hasta ahora. Después se mostrará la estructura de datos y los errores de las DrCadenas.

```
 1. Program EjemploB2;
 2. Var a,b,c,d,e,f,g,h:integer;
 3.
 4. Function Uno(x,y:integer):integer;
 5.          Begin
 6.            if (x<1) then x:=4
 7.            else x:= y+2;
 8.            Uno := x-c;
 9.          End;
10.
11. Procedure Dos(x:integer;Var y,z:integer);
12.           Var a,b:integer;
13.           Begin
14.             read(a,x,y);
15.             If (x<y) then
16.               while (z>0) do
17.                 z:= c+ a+ d;
18.             else
19.               while (y<0) do
20.                 z:= z*x;
21.           End;
22.
23. Procedure Tres(Var x,y:integer);
24.           Var a,b:integer;
25.           Begin
26.             b:=x+y;
27.             case a of
28.               1:while x<1 do x:=x+5;
29.               2:if (b<x) then x:=2
30.                    else b:=uno(x,a);
31.               else: dos(b,a,y)
32.             end
33.           End;
34.
35. BEGIN
36.      while (g<0)do read(g,f,h);
37.      if (g<f) then
38.         dos(a,b,c)
```

```
39.      else
40.         repeat
41.            tres(e,h)
42.         until (h>0)
43. END.
```

Como se ve en el ejemplo hay diversos casos que no hemos tratado, como el echo de que dentro de una función se llame a otra (dentro de la rutina 'tres' se llama a 'uno' y a 'dos'), también existen estructuras anidadas, rutinas con parámetros por valor y por referencia, variables globales, variables pasadas como parámetros en diversas posiciones en la misma llamada a una función, etc. Se puede decir que en este ejemplo se ven todos los tipos de problemas que pudiéramos encontrar al leer el código.

Como resultado de la lectura del código se genera la estructura de datos, a continuación se muestra como queda, apareciendo la lista de apariciones debajo de su variable correspondiente, la lista de variables debajo de su función correspondiente, etc:

```
PROGRAMA ejemplob2
        FUNCION main Linea_inicio 35 Linea_fin 43
                VARIABLE a Tipo local Posicion 0
                        Valor ( NumLinea 36 PosLinea 2
                        Valor)* NumLinea 37 PosLinea 2
                        Valor ( NumLinea 38 PosLinea 1
                        Valor R NumLinea 39 PosLinea 2
                        Valor o NumLinea 39 PosLinea 5
                        Valor ( NumLinea 41 PosLinea 1
                        Valor )+ NumLinea 43 PosLinea 1
                        Valor) NumLinea 43 PosLinea 2
                VARIABLE b Tipo local Posicion 0
                        Valor ( NumLinea 36 PosLinea 2
                        Valor )* NumLinea 37 PosLinea 2
                        Valor ( NumLinea 38 PosLinea 1
                        Valor DR(()*oR(R)*) NumLinea 39 PosLinea 3
                        Valor o NumLinea 39 PosLinea 5
                        Valor ( NumLinea 41 PosLinea 1
                        Valor )+ NumLinea 43 PosLinea 1
                        Valor ) NumLinea 43 PosLinea 2
                VARIABLE c Tipo local Posicion 0
                        Valor ( NumLinea 36 PosLinea 2
```

Valor)* NumLinea 37 PosLinea 2
Valor (NumLinea 38 PosLinea 1
Valor (R(RDR)*o(RD)*) NumLinea 39 PosLinea 4
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor (()*o(o(o)R)o((R)*o()*)) NumLinea 41 PosLinea 2
Valor)+ NumLinea 43 PosLinea 1
Valor) NumLinea 43 PosLinea 2
VARIABLE d Tipo local Posicion 0
Valor (NumLinea 36 PosLinea 2
Valor)* NumLinea 37 PosLinea 2
Valor (NumLinea 38 PosLinea 1
Valor ((R)*o()*) NumLinea 39 PosLinea 1
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor (()*o(o)o((R)*o()*)) NumLinea 41 PosLinea 3
Valor)+ NumLinea 43 PosLinea 1
Valor) NumLinea 43 PosLinea 2
VARIABLE e Tipo local Posicion 0
Valor (NumLinea 36 PosLinea 2
Valor)* NumLinea 37 PosLinea 2
Valor (NumLinea 38 PosLinea 1
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor R(R(RDR)*oR(DoR)o) NumLinea 41 PosLinea 4
Valor)+ NumLinea 43 PosLinea 1
Valor) NumLinea 43 PosLinea 2
VARIABLE f Tipo local Posicion 0
Valor (NumLinea 36 PosLinea 2
Valor D NumLinea 36 PosLinea 4
Valor)* NumLinea 37 PosLinea 2
Valor R NumLinea 37 PosLinea 4
Valor (NumLinea 38 PosLinea 1
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor)+ NumLinea 43 PosLinea 1
Valor) NumLinea 43 PosLinea 2
VARIABLE g Tipo local Posicion 0
Valor R NumLinea 36 PosLinea 1
Valor (NumLinea 36 PosLinea 2
Valor D NumLinea 36 PosLinea 3
Valor R NumLinea 37 PosLinea 1
Valor)* NumLinea 37 PosLinea 2
Valor R NumLinea 37 PosLinea 3
Valor (NumLinea 38 PosLinea 1
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor)+ NumLinea 43 PosLinea 1

Valor) NumLinea 43 PosLinea 2
VARIABLE h Tipo local Posicion 0
Valor (NumLinea 36 PosLinea 2
Valor D NumLinea 36 PosLinea 5
Valor)* NumLinea 37 PosLinea 2
Valor (NumLinea 38 PosLinea 1
Valor o NumLinea 39 PosLinea 5
Valor (NumLinea 41 PosLinea 1
Valor R(()*o(o)o(R(DR)*o(RD)*)) NumLinea 41 PosLinea 5
Valor R NumLinea 42 PosLinea 1
Valor)+ NumLinea 43 PosLinea 1
Valor) NumLinea 43 PosLinea 2
FUNCION uno Linea_inicio 4 Linea_fin 9
VARIABLE x Tipo valor Posicion 1
Valor R NumLinea 6 PosLinea 1
Valor (NumLinea 6 PosLinea 2
Valor D NumLinea 7 PosLinea 1
Valor o NumLinea 7 PosLinea 2
Valor D NumLinea 7 PosLinea 4
Valor) NumLinea 8 PosLinea 1
Valor R NumLinea 8 PosLinea 2
VARIABLE y Tipo valor Posicion 2
Valor (NumLinea 6 PosLinea 2
Valor o NumLinea 7 PosLinea 2
Valor R NumLinea 7 PosLinea 3
Valor) NumLinea 8 PosLinea 1
FUNCION dos Linea_inicio 11 Linea_fin 21
VARIABLE x Tipo valor Posicion 1
Valor D NumLinea 14 PosLinea 2
Valor R NumLinea 15 PosLinea 1
Valor (NumLinea 16 PosLinea 1
Valor (NumLinea 17 PosLinea 1
Valor)* NumLinea 18 PosLinea 2
Valor o NumLinea 18 PosLinea 3
Valor (NumLinea 20 PosLinea 1
Valor R NumLinea 20 PosLinea 3
Valor)* NumLinea 21 PosLinea 2
Valor) NumLinea 21 PosLinea 3
VARIABLE y Tipo referencia Posicion 2
Valor D NumLinea 14 PosLinea 3
Valor R NumLinea 15 PosLinea 2
Valor (NumLinea 16 PosLinea 1
Valor (NumLinea 17 PosLinea 1
Valor)* NumLinea 18 PosLinea 2
Valor o NumLinea 18 PosLinea 3
Valor R NumLinea 19 PosLinea 1
Valor (NumLinea 20 PosLinea 1
Valor R NumLinea 21 PosLinea 1

Valor)* NumLinea 21 PosLinea 2
Valor) NumLinea 21 PosLinea 3
VARIABLE z Tipo referencia Posicion 3
Valor (NumLinea 16 PosLinea 1
Valor R NumLinea 16 PosLinea 2
Valor (NumLinea 17 PosLinea 1
Valor D NumLinea 17 PosLinea 5
Valor R NumLinea 18 PosLinea 1
Valor)* NumLinea 18 PosLinea 2
Valor o NumLinea 18 PosLinea 3
Valor (NumLinea 20 PosLinea 1
Valor R NumLinea 20 PosLinea 2
Valor D NumLinea 20 PosLinea 4
Valor)* NumLinea 21 PosLinea 2
Valor) NumLinea 21 PosLinea 3
VARIABLE a Tipo local Posicion 0
Valor D NumLinea 14 PosLinea 1
Valor (NumLinea 16 PosLinea 1
Valor (NumLinea 17 PosLinea 1
Valor R NumLinea 17 PosLinea 3
Valor)* NumLinea 18 PosLinea 2
Valor o NumLinea 18 PosLinea 3
Valor (NumLinea 20 PosLinea 1
Valor)* NumLinea 21 PosLinea 2
Valor) NumLinea 21 PosLinea 3
VARIABLE b Tipo local Posicion 0
Valor (NumLinea 16 PosLinea 1
Valor (NumLinea 17 PosLinea 1
Valor)* NumLinea 18 PosLinea 2
Valor o NumLinea 18 PosLinea 3
Valor (NumLinea 20 PosLinea 1
Valor)* NumLinea 21 PosLinea 2
Valor) NumLinea 21 PosLinea 3
FUNCION tres Linea_inicio 23 Linea_fin 33
VARIABLE x Tipo referencia Posicion 1
Valor R NumLinea 26 PosLinea 1
Valor (NumLinea 28 PosLinea 1
Valor R NumLinea 28 PosLinea 2
Valor (NumLinea 28 PosLinea 3
Valor R NumLinea 28 PosLinea 4
Valor D NumLinea 28 PosLinea 5
Valor R NumLinea 29 PosLinea 1
Valor)* NumLinea 29 PosLinea 2
Valor o NumLinea 29 PosLinea 3
Valor R NumLinea 29 PosLinea 5
Valor (NumLinea 29 PosLinea 6
Valor D NumLinea 30 PosLinea 1
Valor o NumLinea 30 PosLinea 2

Valor R NumLinea 30 PosLinea 4
Valor) NumLinea 31 PosLinea 1
Valor o NumLinea 31 PosLinea 2
Valor) NumLinea 33 PosLinea 1
VARIABLE y Tipo referencia Posicion 2
Valor R NumLinea 26 PosLinea 2
Valor (NumLinea 28 PosLinea 1
Valor (NumLinea 28 PosLinea 3
Valor)* NumLinea 29 PosLinea 2
Valor o NumLinea 29 PosLinea 3
Valor (NumLinea 29 PosLinea 6
Valor o NumLinea 30 PosLinea 2
Valor) NumLinea 31 PosLinea 1
Valor o NumLinea 31 PosLinea 2
Valor (R(DR)*o(RD)*) NumLinea 32 PosLinea 5
Valor) NumLinea 33 PosLinea 1
VARIABLE a Tipo local Posicion 0
Valor R NumLinea 27 PosLinea 1
Valor (NumLinea 28 PosLinea 1
Valor (NumLinea 28 PosLinea 3
Valor)* NumLinea 29 PosLinea 2
Valor o NumLinea 29 PosLinea 3
Valor (NumLinea 29 PosLinea 6
Valor o NumLinea 30 PosLinea 2
Valor R NumLinea 30 PosLinea 5
Valor) NumLinea 31 PosLinea 1
Valor o NumLinea 31 PosLinea 2
Valor DR(()*oR(R)*) NumLinea 32 PosLinea 4
Valor) NumLinea 33 PosLinea 1
VARIABLE b Tipo local Posicion 0
Valor D NumLinea 26 PosLinea 3
Valor (NumLinea 28 PosLinea 1
Valor (NumLinea 28 PosLinea 3
Valor)* NumLinea 29 PosLinea 2
Valor o NumLinea 29 PosLinea 3
Valor R NumLinea 29 PosLinea 4
Valor (NumLinea 29 PosLinea 6
Valor o NumLinea 30 PosLinea 2
Valor D NumLinea 30 PosLinea 6
Valor) NumLinea 31 PosLinea 1
Valor o NumLinea 31 PosLinea 2
Valor R NumLinea 32 PosLinea 3
Valor) NumLinea 33 PosLinea 1

De este modo se muestra tanto la lista de variables como la lista de apariciones de cada variable; en este caso la lista de funciones se compone de cuatro nodos, el del programa principal (llamado MAIN) y los de las tres rutinas usadas ('Uno', 'Dos' y 'Tres').

Las DrCadenas generadas por cada una de variables una vez analizadas se mostrarían:

- Función: Main
  - Variable : a Tipo : Local

    DrCadena: (**R**o1)

    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 39)
  - Variable : b Tipo : Local

    DrCadena: (DR(1oRR*)o1)

    Errores:
    - No hay errores
  - Variable : c Tipo : Local

    DrCadena: ((**R**(RDR)*o(**R**D)*)o((1o(1o**R**)o(**R***o1)))+)

    Errores:
    - Error por referencia inicial en variable local, error en la R número 1 (línea 39)
    - Error por referencia inicial en variable local, error en la R número 4 (línea 39)
    - Error por referencia inicial en variable local, error en la R número 5 (línea 41)
    - Error por referencia inicial en variable local, error en la R número 6 (línea 41)
  - Variable : d Tipo : Local

    DrCadena: ((**R***o1)o((1o1o(**R***o1)))+)

    Errores:

- Error por referencia inicial en variable local, error en la R número 1 (línea 39)
- Error por referencia inicial en variable local, error en la R número 2 (línea 41)

- Variable : e Tipo : Local

  DrCadena: (1o(**R**(R(RDR)*oR(DoR)o1))+)

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 41)

- Variable : f Tipo : Local

  DrCadena: D*R

  Errores:

  - No hay errores

- Variable : g Tipo : Local

  DrCadena: **R**(DR)*R

  Errores:

  - Error por referencia inicial en variable local, error en la R número 1 (línea 36)

- Variable : h Tipo : Local

  DrCadena: D*(1o(R(1o1o(R(DR)*o(RD)*))R)+)

  Errores:

  - No hay errores

♦ Función: Uno

- Variable : x Tipo : Valor

  DrCadena: R(DoD)R

  Errores:

  - No hay errores

- Variable : y Tipo : Valor

  DrCadena: (1oR)

  Errores:

  - No hay errores

- Variable : c Tipo : Global

  DrCadena: R

  Errores:

  - No se analizan las globales

♦ Función: Dos

- Variable : x Tipo : Valor

  DrCadena: DR(1oR*)

  Errores:

  - Error por no existencia de referencia inicial o final en variable pasada por valor

- Variable : y Tipo : Referencia

  DrCadena: DR(1oRR*)

  Errores:

  - No hay errores

- Variable : z Tipo : Referencia

  DrCadena: (R(DR)*o(RD)*)

  Errores:

  - No hay errores

- Variable : a Tipo : Local

  DrCadena: D(R*o1)

  Errores:

  - No hay errores

- Variable : b Tipo : Local

  DrCadena:

  Errores:

  - Error por no utilización de la variable

- Variable : c Tipo : Global

  DrCadena: (R*o1)

  Errores:

  - No se analizan las globales

- Variable : d Tipo : Global

DrCadena: (R*o1)

Errores:

- No se analizan las globales

- Función: Tres
  - Variable : x Tipo : Referencia

    DrCadena: R(R(RDR)*oR(DoR)o1)

    Errores:

    - No hay errores
  - Variable : y Tipo : Referencia

    DrCadena: R(1o1o(R(DR)*o(RD)*))

    Errores:

    - No hay errores
  - Variable : a Tipo : Local

    DrCadena: **R**(1o(1oR)oDR(1oRR*))

    Errores:

    - Error por referencia inicial en variable local, error en la R número 1 (línea 27)
  - Variable : b Tipo : Local

    DrCadena: D(1oR(1o**D**)oR)

    Errores:

    - Error por definición final en variable local, error en la D número 2 (línea 30)
  - Variable : c Tipo : Global

    DrCadena: (1o(1oR)o(R*o1))

    Errores:

    - No se analizan las globales
  - Variable : d Tipo : Global

    DrCadena: (1o1o(R*o1))

    Errores:
  - No se analizan las globales

7. CONCLUSIONES

De todo el trabajo realizado en este proyecto, se han obtenido bastante resultados y conclusiones, como se puede apreciar a continuación:

- Se ha estudiado:
  - La técnica de análisis de flujo de datos presenta tres tipos de errores esenciales (Figura 2.1), entre los que se encuentran los producidos por una iniciación equivocada de la DR-Cadena (definición inicial no existente), dos definiciones consecutivas y una variable inútilmente definida al final.
  - Pero esto no siempre es así para todas las variables del programa. En el caso de tratar con parámetros (Figura 2.2), se tendrá en cuenta que si son parámetros de salida o de entrada.
  - El paso múltiple de parámetros a una función, tanto por valor como por referencia.
  - Como tratar las variables globales en funciones.
  - Las llamadas a las funciones tanto en el programa principal como dentro de las rutinas.

- Se ha diseñado, usando la orientación a objetos, una estructura para almacenar toda la información necesaria sobre el programa de prueba a estudiar y para poderla analizar.

- Se ha diseñado, usando la orientación a objetos, una estructura para almacenar toda la información necesaria sobre los parámetros cuando se encuentra la llamada a una funcón dentro del código a analizar.

- Se han implementado:
  - Funciones que permiten estudiar estructuras de todo tipo, incluso en el caso de que haya varias anidadas.

- Funciones que permiten estudiar las llamadas a las rutinas, incluso dentro de otras rutinas.
- Funciones que permiten estudiar el paso de parámetros, tanto por valor como por referencia.
- Funciones que permiten que el paso de un parámetro en múltiples posiciones (por valor y por referencia).
- Funciones que estudian el uso de variables globales en funciones.
- La posibilidad de utilizar una variable global en una función, como se ha dicho en el anterior caso, y a la vez llamar a dicha función pasando como parámetro la misma variable global.
- Funciones que estudian las DrCadenas generadas por las varaibles e identifican los errores en el caso de que los tengan.

Se puede finalizar con la siguiente conclusión: la prueba no engloba todas las actividades relativas a la garantía de la calidad, pues no se ocupa de las diferentes etapas que han precedido al desarrollo del programa, pero siempre minimiza las probabilidades de aparición de una anomalía. La prueba es un proceso incompleto que forma parte del proceso de desarrollo y se interesa únicamente por el funcionamiento correcto del producto final de la actividad de los programadores.

8. DESARROLLOS POSTERIORES

Las propuestas que se ofrecen para la continuación de este proyecto son las siguientes:

- Tratamiento de las constantes y de los valores devueltos por las funciones; ya que esto no ha sido tratado en este proyecto y es posible realizar un seguimiento y un análisis, como en el resto de las variables, de las DrCadenas generadas por estos dos elementos.

- Convertir la técnica estática de análisis del flujo de datos en una técnica de aproximación dinámica incluyendo en el proyecto el tratamiento de las variables tipo puntero y de las variables tipo tabla, tanto de una como de varias dimensiones. Esta es una posibilidad que no ha sido contemplada en este trabajo, pero que sí es posible y que ha sido descrita y desarrollada en otros proyectos de investigación.

- Diseñar e implementar los algoritmos necesarios para la búsqueda de bucles infinitos en el código fuente estudiado. Para ello tan solo habría que añadir las funciones necesarias para su identificación, puesto que las funciones de lectura del código ya están implementadas completamente. Este es un caso que se contempla dentro del análisis del flujo de datos, aunque no se pueden desarrollar unos algoritmos infalibles, pero que en este trabajo se ha obviado puesto que este proyecto está más orientado al análisis de las DrCadenas.

- Incluir la posibilidad de leer funciones recursivas dentro del código fuente facilitado, para la cual sería incluso necesario modificar la estructura de datos usada en este proyecto, ya que esta se basa en el principio de que toda función llamada dentro del código ha sido previamente definida al completo.

- Incluir el tratamiento de las unidades del lenguaje de programación Pascal. En este trabajo no se tienen en cuenta ya que se obvia su inclusión y de igual modo no se leen variables o funciones que no se hayan definido dentro del código fuente facilitado. Se podrían implementar los algoritmos necesarios para tratarlas y analizarlas antes de comenzar a leer el código principal.

- Incluir el tratamiento de un mayor número de funciones del compilador, ya que en este proyecto solo se tienen en cuenta las consideradas principales (*read, readln, write, writeln, inc*, etc...). Todas aquellas funciones que la aplicación no considere como del compilador y que no estén definidas previamente no serán leídas ni tratadas. Se podrían incluir los algoritmos necesarios para que el programa tratase un número mucho mayor de estas funciones.

- Implementar una interfaz más visual y completa para la aplicación ya que, por ejemplo, la actual no permite un estudio independiente de cada variable, puesto que cada vez que el programa se ejecuta muestra todas las variables existentes de forma ordenada. Una interfaz más gráfica e intuitiva podría simplificar el manejo de la aplicación por parte del usuario.